

Elisabeth Fughe
Matrikelnummer: 5263769
s3499227@stud.uni-frankfurt.de

Bachelorarbeit (B.Sc. - Informatik)

**Können Modelle der Ökonophysik
Preis-Dynamiken vorhersagen?**

**Fitting von Markt-Modellen auf empirischen Daten mittels Bayesscher
Statistik**

Elisabeth Fughe

Abgabedatum: 16. April 2019

FIAS - Frankfurt Institute for Advanced Studies
Prof. Dr. Nils Bertschinger

Zusammenfassung

Inhaltsverzeichnis

1. Einleitung	5
2. Verwendete Methoden	6
2.1. Bayessche Modellierung	6
2.2. Markov Chain Monte Carlo (MCMC)	7
2.3. Hamiltonian Monte Carlo Sampling (HMC)	10
2.4. <i>Stan</i> und <i>R</i>	13
3. Die Modelle	17
3.1. Generalized Autoregressive Conditional Heteroscedasticity (GARCH)	17
3.2. Franke & Westerhoff (FW)	21
3.3. Alfarano, Lux & Wagner (AL)	28
4. Simulationen	34
4.1. Daten & Vorhersagen	34
4.2. Ergebnisse	35
Anhang	43
A. <i>Stan</i> Code	43
B. <i>R</i> Code	50
C. Weitere Abbildungen	61
Eigenständigkeitserklärung	81

Abbildungsverzeichnis

2.1.	Verhalten der Markov-Kette beim Ziehen von Stichproben [4]	7
2.2.	Konzentration der Markov-Kette auf die typische Menge einer Verteilung [4]	8
2.3.	Starke Krümmungen (grün) der typischen Menge einer Verteilung führen zu verzerrten MCMC-Schätzern [4]	9
2.4.	Mit Hilfe eines Vektorfeldes das an der typischen Menge ausgerichtet ist, kann die Ziel-Verteilung leichter abgetastet werden [4].	10
2.5.	Die Richtung des Gradienten zeigt immer von der typischen Menge in Richtung parameter-sensitiver Bereiche(grüner Pfeil), wie z.B. dem Modus der Verteilung. Folgt der Sampler diesen Richtungen, kann er die typische Menge nicht erkunden [4].	11
2.6.	Analoges System der klassischen Physik [4].	12
3.1.	Die Daten weisen im Zeitverlauf unterschiedliche Abweichung zur Trendgerade auf, d.h. die Residuen/Störterme sind nicht konstant im Zeitverlauf und es liegt Heteroskedastizität vor (Eigene Darstellung).	18
3.2.	Die Studentsche t-Verteilung mit 5 Freiheitsgraden, dem Erwartungswert 0 skaliert um den Faktor 1 (Eigene Darstellung).	25
4.1.	Traceplots der Chains des FW-Modells(ma) auf den Daten der Dotcom-Blase für August 2008 mit 381 divergent Transitions und 8 Transitions über <code>max_treedepth = 18</code> wobei <code>adapt_delta = 0.9</code> (eigene Darstellung)	36
4.2.	Traceplots der Chains des FW-Modells (ma) auf den Daten der Finanzkrise für Juli 2008 mit 0 divergent Transitions und 0 Transitions über <code>max_treedepth = 18</code> wobei <code>adapt_delta = 0.99</code> (eigene Darstellung)	36
4.3.	Traceplots der Chains des FW-Modells (ma) auf den Daten der Finanzkrise für November 2008 mit 0 divergent Transitions und 0 Transitions über <code>max_treedepth = 29</code> wobei <code>adapt_delta = 0.99</code> (eigene Darstellung)	37
4.5.	Traceplot der Chains des FW-Modells (walk) auf den Daten der Finanzkrise für Dezember 2008 mit 0 divergent Transitions und 0 Transitions über <code>max_treedepth = 128</code> wobei <code>adapt_delta = 0.99</code> (eigene Darstellung)	37
4.4.	Parallele-Koordinaten-Plot der Chains des FW-Modells (ma) auf den Daten der Finanzkrise für November 2008 mit 0 divergent Transitions und 0 Transitions über <code>max_treedepth = 29</code> wobei <code>adapt_delta = 0.99</code> (eigene Darstellung)	38
4.6.	Preis-Vorhersage der Chains des FW-Modells (walk) auf den Daten der Finanzkrise für Dezember 2008 mit 0 divergent Transitions und 0 Transitions über <code>max_treedepth = 128</code> wobei <code>adapt_delta = 0.99</code> (eigene Darstellung)	38

4.7. Preis-Vorhersage der Chains des AL-Modells auf den Daten der Dotcom-Blase für September 2000 mit 2 divergent Transitions und 0 Transitions über <code>max_treedepth = 16</code> wobei <code>adapt_delta = 0.92</code> (eigene Darstellung)	39
4.8. Traceplots der Chains des AL-Modells auf den Daten der Dotcom-Blase für September 2000 mit 2 divergent Transitions und 0 Transitions über <code>max_treedepth = 16</code> wobei <code>adapt_delta = 0.92</code> (eigene Darstellung)	40
4.9. Autokorrelation der Modell-Parameter innerhalb der Chains des AL-Modells auf den Daten der Dotcom-Blase für Juni 2000 mit 1 divergent Transitions und 0 Transitions über <code>max_treedepth = 16</code> wobei <code>adapt_delta = 0.9</code> (eigene Darstellung)	41
4.10. Autokorrelation der Modell-Parameter innerhalb der Chains des AL-Modells auf den Daten der Finanzkrise für Dezember 2008 mit 4 divergent Transitions und 0 Transitions über <code>max_treedepth = 13</code> wobei <code>adapt_delta = 0.85</code> (eigene Darstellung)	41
4.11. Preis-Vorhersage der Chains des GARCH-Modells auf den Daten der Dotcom-Blase für Januar 2000 mit 0 divergent Transitions und 0 Transitions über <code>max_treedepth = 10</code> wobei <code>adapt_delta = 0.8</code> (eigene Darstellung)	41
C.1.1. Übersicht über die Anzahl der divergent Transitions je Monat aller Modelle auf den verschiedenen Daten (eigene Darstellung)	61
C.1.2. Übersicht über die Anzahl der Iterationen, die die <code>max_treedepth</code> überschritten haben je Monat, Modell und Datensatz (eigene Darstellung)	62
C.1.3. Übersicht über den verwendeten Wert des Parameters <code>adapt_delta</code> in der Simulation je Monat, Modell und Datensatz (eigene Darstellung)	62
C.1.4. Übersicht über den verwendeten Wert des Parameters <code>max_treedepth</code> in der Simulation je Monat, Modell und Datensatz (eigene Darstellung)	63
C.1.5. Übersicht über die durchschnittlich benötigte Zeit des Samplers zur Simulation je Monat, Modell und Datensatz (eigene Darstellung)	63
C.2.1. Preis-Vorhersagen des GARCH-Modells auf den Daten der Dotcom-Blase . .	65
C.2.2. Preis-Vorhersagen des FW-Modells (ma) auf den Daten der Dotcom-Blase . .	66
C.2.3. Preis-Vorhersagen des FW-Modells (walk) auf den Daten der Dotcom-Blase .	67
C.2.4. Preis-Vorhersagen des AL-Modells auf den Daten der Dotcom-Blase	68
C.2.5. Preis-Vorhersagen des GARCH-Modells auf den Daten der Finanzkrise . . .	69
C.2.6. Preis-Vorhersagen des FW-Modells (ma) auf den Daten der Finanzkrise . . .	70
C.2.7. Preis-Vorhersagen des FW-Modells (walk) auf den Daten der Finanzkrise . .	71
C.2.8. Preis-Vorhersagen des AL-Modells auf den Daten der Finanzkrise	72
C.3.1. Preis-Vorhersagen des GARCH-Modells auf den Daten der Dotcom-Blase . .	73
C.3.2. Preis-Vorhersagen des FW-Modells (ma) auf den Daten der Dotcom-Blase . .	74
C.3.3. Preis-Vorhersagen des FW-Modells (walk) auf den Daten der Dotcom-Blase .	75
C.3.4. Preis-Vorhersagen des AL-Modells auf den Daten der Dotcom-Blase	76
C.3.5. Preis-Vorhersagen des GARCH-Modells auf den Daten der Finanzkrise . . .	77
C.3.6. Preis-Vorhersagen des FW-Modells (ma) auf den Daten der Finanzkrise . . .	78
C.3.7. Preis-Vorhersagen des FW-Modells (walk) auf den Daten der Finanzkrise . .	79
C.3.8. Preis-Vorhersagen des AL-Modells auf den Daten der Finanzkrise	80

1. Einleitung

2. Verwendete Methoden

In den nachfolgenden Kapiteln 2.1 bis 2.4 werden die mathematischen Grundlagen, die verwendeten Methoden sowie die Programmiersprache erläutert. Um verschiedene Modelle auf den gleichen Daten anzuwenden und vergleichen zu können, wurden die Modelle von Bertschinger, Mozzhorin und Sinha [3] in der Programmiersprache *Stan* beschrieben. Anschließend ist das Fitten der Modelle auf den Daten mittels R leicht umsetzbar.

2.1. Bayessche Modellierung

Die Bayessche Statistik untersucht mittels Bayesscher Wahrscheinlichkeiten und dem Satz von Bayes Fragestellungen der Stochastik. Anders als in der klassischen Statistik, die unendlich oft wiederholbare Zufallsexperimente voraussetzt, steht die Verwendung und Modellierung von Wahrscheinlichkeitsverteilungen im Vordergrund.

Es gilt, dass beobachtete Daten

$$x = (x_1, \dots, x_n)$$

mittels bedingter Wahrscheinlichkeiten in Beziehung zu unbekannten Parametern

$$\theta = (\theta_1, \dots, \theta_m)$$

stehen. So kann die gemeinsame Wahrscheinlichkeitsdichte

$$p(x, \theta) = p(x|\theta) \cdot p(\theta)$$

durch die a-priori-Verteilung unbekannter Parameter $p(\theta)$ und den Erkenntnissen aus dem Datensatz $p(x|\theta)$ berechnet werden. Mit Hilfe des Satzes von Bayes kann dann die a-posteriori-Verteilung unbekannter Parameter

$$p(\theta|x) = \frac{p(x|\theta) \cdot p(\theta)}{p(x)}$$

ermittelt werden [3].

Die a-posteriori-Verteilung enthält somit Informationen über die unbekannten Parameter durch die Kombination der a-priori Verteilung mit den Informationen, die aus den beobachteten Daten gewonnen wurden. Sie wird zur Punktschätzung und zur Schätzung von Konfidenzintervallen genutzt und daher auch oft als Ziel-Verteilung bezeichnet.

So sind Bayessche Modelle im Gegensatz zur klassischen Statistik auf kleineren Datensätzen anwendbar, dort ergibt sich jedoch eine breite Wahrscheinlichkeitsverteilung, die somit unter Umständen eine geringe Genauigkeit aufweist.

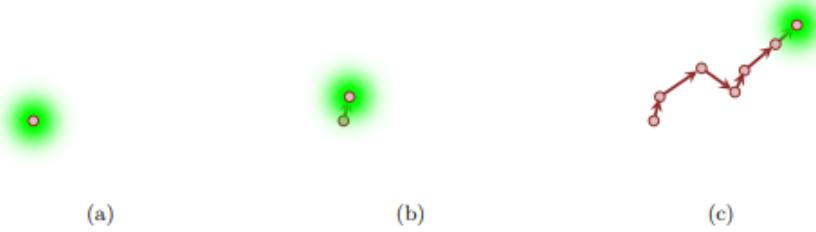


Abbildung 2.1.: Verhalten der Markov-Kette beim Ziehen von Stichproben [4]

2.2. Markov Chain Monte Carlo (MCMC)

In der Bayesschen Statistik beschreibt die a-posteriori-Verteilung die Unsicherheit der unbekannten Parameter, die anhand beobachteter Daten geschätzt wurden. Mit der Markov Chain Monte Carlo (MCMC) Methode können die a-posteriori-Verteilung und dadurch die unbekannten Parameter untersucht werden [11].

Um aus der Ziel-Verteilung Stichproben ziehen zu können, wird eine Markov-Kette entworfen, deren stationäre Verteilung der Ziel-Verteilung entspricht. Hier wird eine Stichprobe aus der a-posteriori-Wahrscheinlichkeitsdichte benötigt, sodass das langfristige Gleichgewicht der Markov-Kette der a-posteriori-Wahrscheinlichkeitsdichte entspricht. Der Zustand der Markov-Kette, der sich möglichst nah am Gleichgewicht befindet, entspricht dann einem Element der Stichprobe. Es gilt somit, dass sich die Qualität der Stichprobe mit der Anzahl der Schritte, die die Markov-Kette zurücklegt, verbessert.

Zur Konstruktion der Markov-Kette wird eine Übergangsfunktion berechnet, die die Ziel-Wahrscheinlichkeitsdichte invariant lässt, damit die Ziel-Dichte der stationären Verteilung der Markov-Kette entspricht. Sei z.B. $p(\theta)$ die Ziel-Dichte und sei $t(\theta'|\theta)$ die Übergangsdichte, dann gilt

$$p(\theta') = \int t(\theta'|\theta) \cdot p(\theta) d\theta$$

Stellt man sich die Markov-Kette als zufällig durch einen Parameterraum, also z.B. die a-posteriori-Verteilung, wandernden Punkt vor, so entspricht die n -te Komponente der Verteilung der relative Wahrscheinlichkeit, den Punkt im Zustand n anzutreffen.

Die Übergangsdichte (grün) (a) in Abbildung 2.1 beschreibt die Wahrscheinlichkeit eines neuen Punktes im Parameterraum in Abhängigkeit der aktuellen Position d.h. die Wahrscheinlichkeit des nächsten Schrittes der Markov-Kette auf ihrem Weg durch die Verteilung in Abhängigkeit ihrer aktuellen Position. Werden Stichproben aus dieser Verteilung gezogen, also wurden zufällig gewählte nächste Schritte vom Algorithmus akzeptiert, ergibt sich ein weiterer Zustand in der Markov-Kette und eine neue Verteilung, von der Stichproben gezogen werden können (vgl. (b) in Abbildung 2.1). So wandert die Markov-Kette durch den Parameterraum (c) (vgl. Abbildung 2.1).



Abbildung 2.2.: Konzentration der Markov-Kette auf die typische Menge einer Verteilung [4]

Wenn die Übergangsdichte die Ziel-Verteilung beibehalten soll, konzentriert sie sich auf deren typische Menge¹ (rot) siehe Abbildung 2.2. Unter idealen Bedingungen entdeckt die Markov-Kette in drei verschiedenen Phasen:

1. Die Markov-Kette konvergiert gegen die typische Menge der Verteilung. Die MCMC-Schätzer sind stark verzerrt.
2. Die Markov-Kette hat die typische Menge erkannt und verweilt dort. Diese erste Erkundung der typischen Menge ist äußerst effektiv und verringert die Verzerrung der MCMC-Schätzer erheblich, da die Verzerrung der ersten Stichproben eliminiert wird.
3. In der dritten Phase wird die Präzision der MCMC-Schätzer durch weiteres Erforschen der typischen Menge weiter verfeinert, so dass sie den zentralen Grenzwertsatz erfüllen. Das bedeutet, dass die durch die Markov-Kette gezogenen Stichproben asymptotisch der Ziel-Verteilung folgen.

Ideale Bedingungen liegen allerdings nicht vor, wenn die typische Menge der a-posteriori-Verteilung z.B. eine starke Krümmung aufweist (siehe Abbildung 2.3). Die Markov-Kette kann damit nicht umgehen und aus diesem stark gekrümmten Bereich keine Stichprobe ziehen und ignoriert diesen einfach und verzerrt dadurch die MCMC-Schätzer. Da Markov-Ketten die exakten Erwartungen asymptotisch wiederherstellen müssen, ist eine Kompensation dafür nötig, dass manche Regionen der Ziel-Verteilung nicht beachtet wurden. Dazu bleibt die Markov-Kette an der Grenze der Krümmung hängen. So werden die MCMC-Schätzer annähernd so berechnet, als würde die Markov-Kette den gekrümmten Bereich erkunden. Allerdings erhält man so überschätzte MCMC-Schätzer. Eine starke Krümmung in der typischen Menge der Ziel-Verteilung führt also zu verzerrten MCMC-Schätzern, die den zentralen Grenzwertsatz nicht mehr erfüllen [4].

¹**Typical Set (hier: typische Menge):** Die typische Menge ist ein Begriff aus der Informationstheorie und hängt direkt mit dem Begriff der Entropie zusammen. Es gilt, dass die Summe aller Wahrscheinlichkeiten aller Elemente der typischen Menge annähernd 1 ist [18]. Daher enthält die typische Menge eine gute Beschreibung des Parameterraums.

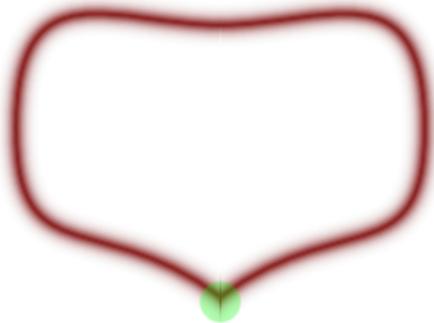


Abbildung 2.3.: Starke Krümmungen (grün) der typischen Menge einer Verteilung führen zu verzerrten MCMC-Schätzern [4]

Ein weit verbreitetes MCMC-Verfahren zur Konzentration auf die typische Menge der zu untersuchenden Verteilung ist der Metropolis-Hastings-Algorithmus. Der Algorithmus startet an einem zufälligen Punkt in der zu untersuchenden Verteilung. Dann wird eine Schrittweite zufällig mit Hilfe einer symmetrischen Wahrscheinlichkeitsverteilung gewählt. Der Schritt wird auf Grundlage der Wahrscheinlichkeit der neuen Position im Verhältnis zur alten Position abgelehnt oder akzeptiert [11]. So wird sicher gestellt, dass die Markov-Kette von jedem Punkt aus gegen das langfristige Gleichgewicht der Ziel-Wahrscheinlichkeitsdichte konvergiert [3].

Der Metropolis-Hastings-Algorithmus ist sehr einfach zu implementieren und liefert gute Ergebnisse, allerdings nicht bei stark korrelierten Parametern [11] und hoher Komplexität der Verteilung [4]. Denn wenn die a-posteriori-Verteilung stark gekrümmmt ist, bleibt die Markov-Kette an diesen Rändern hängen und lehnt so gut wie jeden weiteren Schritt ab. So kann nicht aus allen Bereiche der Ziel-Verteilung Stichproben gezogen werden und es entsteht eine verzerrte Stichprobe.

Je mehr Dimensionen die Verteilung hat, desto kleiner wird die typische Menge der Verteilung. Das führt dazu, dass fast immer Punkte außerhalb der typischen Menge gewählt werden. Die Dichte des Punktes ist so klein, dass der Schritt vom Metropolis-Hastings-Algorithmus immer abgelehnt wird. Das führt dazu, dass sich die Markov-Kette kaum weiter bewegt. Erhöht man die Akzeptanz, indem man die Anzahl der Punkte reduziert, die sich in der typischen Menge befinden müssen, führt das dazu, dass die Markov-Kette extrem langsam konvergiert [4]. In der Theorie würde die Markov-Kette das langfristige Gleichgewicht erreichen, aber in der Praxis stehen dazu nicht unendliche Ressourcen zur Berechnung zur Verfügung.

Der Metropolis-Hastings-Algorithmus kann auch als stochastisches Optimierungsverfahren verstanden werden. Mittels Simulated Annealing² nähert man sich mit immer höherer Wahrscheinlichkeit an ein Minimum an. Ist die Krümmung der Verteilung zu stark, kann der Algorithmus das lokale Minimum nicht mehr verlassen und so wird fast jeder weitere Schritt der

²Simulated Annealing: „Ein heuristisches Approximationsverfahren. Es wird zum Auffinden einer Näherungslösung von Optimierungsproblemen eingesetzt, die durch ihre hohe Komplexität das vollständige Ausprobieren aller Möglichkeiten und mathematische Optimierungsverfahren ausschließen“[17].

Markov-Kette abgelehnt. Letztlich wird der Bereich der Verteilung ignoriert bzw. die Markov-Kette bleibt am Rand des unerwarteten Bereichs hängen. Das führt zu einer Verzerrung der resultierenden MCMC-Schätzer [4].

2.3. Hamiltonian Monte Carlo Sampling (HMC)

In Modellen mit vielen Dimensionen ist es möglich, dass die Markov-Kette mittels Metropolis-Hastings-Algorithmus nicht alle Bereiche der Ziel-Verteilung in einer angemessenen Zeit abtasten kann. Die Strategie des Metropolis-Algorithmus, zufällige Schritte zu wählen und danach zu entscheiden, ob der Schritt akzeptiert wird oder nicht, ist für Modelle mit vielen Dimensionen weniger erfolgreich. Denn es existieren exponentiell viele Richtungen, in die sich der Sampler bewegen kann, aber nur ein Bruchteil der Richtungen belässt den Sampler in der typischen Menge und führt letztlich zu akzeptierten Stichproben.

Um große Schritte vom initialen Punkt in unentdeckte Bereiche der Ziel-Verteilung machen zu können, müssen Informationen über die Geometrie der Verteilung genutzt werden. Die Markov-Kette soll Übergänge berechnen, die der Masse mit hohen Wahrscheinlichkeiten folgt, sich also zusammenhängend durch die typische Menge bewegen. Diese Hamiltonian-Markov-Übergänge können durch Nutzung der differenzierbaren Struktur der a-posteriori-Verteilung berechnet werden.

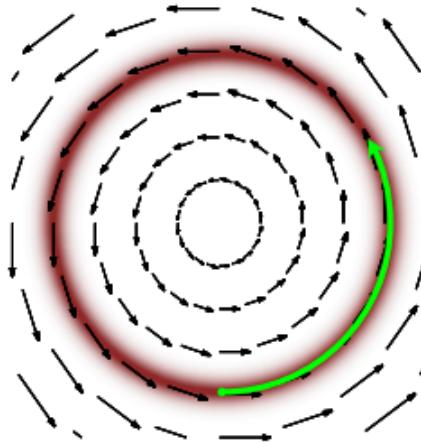


Abbildung 2.4.: Mit Hilfe eines Vektorfeldes das an der typischen Menge ausgerichtet ist, kann die Ziel-Verteilung leichter abgetastet werden [4].

Um die Zielverteilung schneller und gezielter abtasten zu können, wird jedem Punkt im Parameterraum eine Richtung für eine kleine Distanz zugewiesen, z.B. durch ein Vektorfeld, dass an der typischen Menge ausgerichtet ist (vgl. Abbildung 2.4). Folgt der Sampler den Richtungen, entsteht eine zusammenhängende Kurve (*engl. Trajectory*), die effizient und

möglichst schnell vom initialen Punkt durch unentdeckte Bereiche der typischer Menge der Ziel-Verteilung führt [4].

Um jedem Punkt eine Richtung zuzuweisen, kann ein Vektorraum mittels Informationen der Ziel-Verteilung konstruiert werden. Dazu wird die differenzielle geometrische Struktur der a-posteriori-Verteilung benötigt, die wir durch den Gradienten der Ziel-Dichte erhalten können. Denn der Gradient der Ziel-Dichte definiert eben ein solches für die geometrische Struktur der Verteilung sensitives Vektorfeld im Parameterraum.

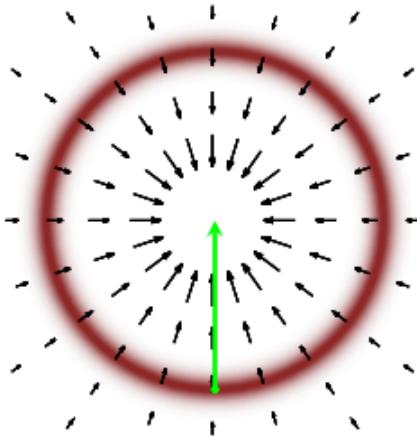


Abbildung 2.5.: Die Richtung des Gradienten zeigt immer von der typischen Menge in Richtung parameter-sensitiver Bereiche(grüner Pfeil), wie z.B. dem Modus der Verteilung. Folgt der Sampler diesen Richtungen, kann er die typische Menge nicht erkunden [4].

Allerdings hängen Ziel-Dichte sowie ihr Gradient stark von den Verteilungsparametern ab, sodass der Gradient immer in Richtung parameter-sensitiver Bereiche wie z.B. dem Modus³ der Verteilung zeigt (vgl. Abbildung 2.5). Das Vektorfeld enthält somit nicht genügend Informationen, um den Sampler durch die typische Menge in parameter-invariante Bereiche zu führen. Um parameter-invariante Bereiche der Ziel-Dichte zu erkunden, muss der Gradient um zusätzliche geometrische Informationen ergänzt werden.

Das Ergänzen der Informationen ist durch Differentialgeometrie möglich, die auch die Mathematik der klassischen Physik beschreibt. Es gilt, dass für jedes probabilistische System ein mathematisch äquivalentes physikalisches System existiert [4]. Das Erkunden der Ziel-Dichte kann äquivalent auch als Erkunden eines physikalischen Systems beschrieben werden. Sei der Modus der Ziel-Dichte ein Planet und der Gradient sein Gravitationsfeld, dann sei die typische Menge der Dichte, der Bereich um den Planeten, durch den ein Satellit kreisen soll (vgl.

³**Modus:** Der Modus ist ein Begriff der deskriptiven Statistik. Er entspricht dem häufigsten Wert einer Stichprobe [16].

Abbildung 2.6).



Abbildung 2.6.: Analoges System der klassischen Physik [4].

Es gilt also das probabilistische Modell so um Impulse (*engl. Momentum*) zu erweitern, dass analog der Satellit die Anziehungskraft des Planeten ausbalanciert und die gewünschte Umlaufbahn nicht verlässt [4]. Der Hamiltonian Monte Carlo Algorithmus ist der einzige Algorithmus, der Impulse mit einer probabilistischen Struktur, die konservative Dynamiken erlaubt, implementiert [4].

Der HMC-Algorithmus zieht also Stichproben aus einem erweiterten Zustandsraum mit Impuls i eines Teilchens an der Position P .

Für die Dichte $d(P, i)$ des Zustandsraums gilt [3]:

$$\begin{aligned} d(P, i) &= d(P)d(i|P) \\ &= e^{\log d(P)+\log d(i|P)} \\ &= e^{-H(P,i)} \end{aligned} \tag{2.1}$$

Wobei die Hamiltonian-Korrektur $H(P, i)$ in Analogie zum physikalischen System der Summe potentieller und kinetischer Energie entspricht.

$$H(P, i) = -\log d(P) - \log d(i|P)$$

Die Hamiltonian Dynamiken \hat{P} und \hat{i}

$$\begin{aligned}\hat{P} &= \frac{\partial}{\partial i} H(P, i) \\ &= -\frac{\partial}{\partial i} \log d(i|P) \\ \hat{i} &= -\frac{\partial}{\partial P} H(P, i) \\ &= \frac{\partial}{\partial P} \log d(P) + \frac{\partial}{\partial P} \log d(i|P)\end{aligned}\tag{2.2}$$

erhalten die gesamte Energie bzw. Wahrscheinlichkeit und so kann beim HMC Sampling stets ein Schritt berechnet werden, der akzeptiert wird [3], sodass die Markov-Kette auch aus stark gekrümmten Bereichen akzeptierte Stichproben ziehen kann. Besonders in Modellen mit vielen Dimensionen schafft das HMC Sampling so wichtige Effizienzvorteile.

Theoretisch ist HMC Sampling unempfindlich bezüglich stark korrelierter Parameter, allerdings werden die Differenzialgleichungen in der Praxis mittels endlicher Schrittweite gelöst. Das kann dazu führen, dass stark gekrümmte Bereiche der Ziel-Dichte doch nicht mehr erreicht werden. Außerdem gilt es die Schrittweite entsprechend der Parameter-Maßeinheiten und -Größen anzupassen [3].

2.4. Stan und R

Stan ist eine Open-Source Plattform für statistische Modellierung und High-Performance Berechnungen. *Stan* ist für alle in der Datenanalyse weit verbreiteten Sprachen (R, Python, shell MATLAB, Julia, Stata) verfügbar und läuft auf den gängigen Betriebssystem (Linux, Mac, Windows) [6].

Stan ist eine höhere Programmiersprache und ermöglicht die Beschreibung der gemeinsamen Verteilung eines Modells auf einer hohen Abstraktionsebene. Die Sprache unterstützt viele Inferenz-Algorithmen, insbesondere Algorithmen, die das Gradientenabstiegsverfahren nutzen, um eine maximale a-posteriori-Schätzung zu erhalten, HMC Sampling sowie Bayesische Gradientenvariation (stochastic gradient variational Bayes). *Stan* Code wird in C++ kompiliert und die hohe Abstraktionsebene der Sprache bietet dem Nutzer einige Vorteile, z.B. müssen bei der Nutzung von HMC Sampling die Gradienten nicht manuell implementiert werden. Benötigte Gradienten werden built-in mittels C++ Library für automatisches Differenzieren berechnet [3].

Ein *Stan*-Programm muss mindestens einen `data`-, einen `Parameter`-Block sowie einen `model`-Block enthalten. Ein Beispiel dafür ist der *Stan*-Code des in dieser Arbeit verwendeten GARCH-Modells:

Im `data`-Block wird definiert, welche Daten benötigt werden, um das Modell zu fitten. Die Variablen in diesem Block entsprechen also den tatsächlichen Beobachtungen und der Block definiert, in welchem Format diese Daten vorliegen.

Hier wird zum Beispiel ein Vektor mit Zeitpunkten T , ein Vektor mit Angabe, ob es sich um eine tatsächliche Beobachtung oder um einen zu vorhersagenden Wert handelt, oder einen Vektor der beobachtete Returns in jedem Zeitpunkt enthält, erwartet.

Im `transformed data` Block können berechnete Daten/Variablen definiert werden. Hier zum Beispiel wird über die Daten iteriert und festgestellt, wie viele Einträge als *Missing Value* markiert wurden. So werden Beobachtungen und Vorhersagen bzw. die zu vorhersagende Daten unterschieden.

```

,
1 data {
2   int<lower=0> T;
3   int<lower=0, upper=1> miss_mask[T];
4   real ret_obs[T]; // Note: Masked indices treated as missing
5 }
6 transformed data {
7   int N = 0; // number of missing values
8   for (t in 1:T)
9     if (miss_mask[t] == 1) N = N + 1;
10 }
```

Daran anschließend kommt der `parameters`-Block, der die unbekannten Parameter des Modells definiert. Anschließend daran kann der `transformed parameters`-Block dazu genutzt werden, um z.B. die Berechnungsvorschrift weiterer Parameter festzulegen.

Hier wird der Return für jeden Zeitpunkt T berechnet: Handelt es sich um eine tatsächliche Beobachtung, wird der beobachtete Return genommen, ansonsten erfolgt die Berechnung mit Hilfe der Standardabweichung im aktuellen Zeitpunkt berechnet:

$\text{ret}[t] = \mu + \sigma[t] * \text{eps_miss}[t]$

```

,
1 parameters {
2   real mu;
3   real<lower=0> alpha0;
4   real<lower=0,upper=1> alphal;
5   real<lower=0,upper=(1-alphal)> betal;
6   real<lower=0> sigma1;
7   real eps_miss[N]; // missing normalized return innovations
8 }
9 transformed parameters {
10   real ret[T]; // returns: observed or r_t=mu+sigma_t*eps_t
11   real<lower=0> sigma[T];
12
13   {
14     int idx = 1; // missing value index
```

```

15 sigma[1] = sigmal;
16 if (miss_mask[1] == 1) {
17     ret[1] = mu + sigma[1] * eps_miss[idx];
18     idx = idx + 1;
19 } else
20     ret[1] = ret_obs[1];
21
22 for (t in 2:T) {
23     sigma[t] = sqrt(alpha0
24         + alphal * pow(ret[t - 1] - mu, 2)
25         + betal * pow(sigma[t - 1], 2));
26     if (miss_mask[t] == 1) {
27         ret[t] = mu + sigma[t] * eps_miss[idx];
28         idx = idx + 1;
29     } else
30         ret[t] = ret_obs[t];
31 }
32 }
33 }
34 }
```

Zum Schluss kommt der `model`-Block, der die Berechnungsvorschrift der Wahrscheinlichkeitsdichte des Modells enthält [6].

Hier die Annahme, dass `mu` und `sigma` standardnormalverteilt sind. Die Returns sind normalverteilt zu den Parametern des Modells (`mu`, `sigma`), wobei es sich um die Standardabweichung in jedem Zeitpunkt T handelt. Weitere Erläuterungen befinden sich im Kapitel 3.1.

Anschließend daran können im `generated quantities`-Block weitere Berechnungen definiert werden, hier z.B. die Berechnungsvorschrift der bedingten gemeinsamen Wahrscheinlichkeitsdichte.

```

,
1 model {
2     mu ~ normal(0, 1);
3     sigmal ~ normal(0, 1);
4
5     ret ~ normal(mu, sigma);
6     // Jacobian correction for transformed innovations
7     for (t in 1:T) {
8         if (miss_mask[t] == 1)
9             target += log(sigma[t]);
10    }
11 }
12 generated quantities {
13     real log_lik[T];
14
15     for (t in 1:T)
16         log_lik[t] = normal_lpdf(ret_obs[t] | mu, sigma[t]);
17 }
```

Der *Stan*-Code der Modelle, die in dieser Arbeit genutzt wurden, befindet sich im Anhang A und wird im Kapitel 3 näher erläutert.

Die Modelle wurden mittels *Rstan* in *R* gefüllt. *Rstan* ermöglicht es *Stan*-Modelle in *R* zu kompilieren, zu testen und zu analysieren. Außerdem wurden die Pakete *tidybayes* und *tidyverse* genutzt, um die Plots zu generieren. Die tidy* Pakete zeichnen sich dadurch aus, dass sie der komplexen Datenstruktur des berechneten Fits Informationen leicht entnehmen können und in einer Datenstruktur zur Verfügung stellen, die anschließend z.B. mittels *ggplot*-Packet leicht in *R* visualisiert werden kann.

Zuerst werden die Daten von der CSV-Datei in eine geeignete Datenstruktur in *R* eingelesen. Anschließend werden Ertrag sowie logarithmierter Ertrag für jeden Zeitpunkt berechnet. Zum Schluss werden Daten mit Dummy-Werten angelegt, die als Vorhersagen dienen und entsprechend als *Missing Value* markiert werden. *data* enthält nun geeignete Strukturen, die als Input für die *Stan*-Modelle dienen können (vgl. *R*-Skript 4 im Anhang B).

Des Weiteren wurden verschiedene kleine *R*-Skripte und -Funktionen geschrieben, um z.B. Stanfit-Objekte aus dem RStudio Workspace zu sichern und wieder zu importieren oder eigene Traceplots der Stanfit-Objekte zu erstellen bzw. Diagnose-Plots einer Liste von Stanfit-Objekten zu erstellen und zu speichern (siehe Anhang B *R*-Skripte 1 bis 5).

3. Die Modelle

In den letzten Jahren haben agentenbasierte Asset-Pricing-Modelle an Bedeutung gewonnen. Im Gegensatz zur Rationalitätsannahme des Homo oeconomicus, werden nur einfache heuristische Strategien über das Verhalten von Marktteilnehmern getroffen. In dieser Arbeit wurden zwei agentenbasierte Modelle auf Finanzmarktdaten des S&P 500 Indexes simuliert. Zum einen das Modell nach Franke und Westerhoff (FW) in zwei Varianten, zum anderen das Modell nach Alfarano, Lux und Wagner (AL). Die zwei Varianten des FW-Modells unterscheiden sich in der Art der Simulation des Fundamentalpreises als gleitenden Durchschnitt(Moving Average) bzw. mittels zufällige Irrfahrt (Random Walk). Die Varianten sind im Kapitel 3.2 näher erläutert. Das AL-Modell wird im Detail in Kapitel 3.3 dokumentiert. Als Vergleichsmodell wurde ein klassisches GARCH(1,1)-Modell gewählt, das in Kapitel 3.1 näher beschrieben wird. Die verwendeten Daten werden in Kapitel 4.1 im Detail erläutert.

Alle Modelle wurden in der Sprache *Stan* so modelliert, dass sie logarithmierte Erträge erwarten und vorhersagen. Des Weiteren wird ein Vektor benötigt, der die Erträge als beobachtet bzw. fehlend(Missing) markiert. Die Anzahl der zu vorhersagenden Zeitpunkte entspricht der Anzahl der als Missing markierten Erträge. Im *Stan*-Code gilt dann entsprechend für jedes Modell der gleiche Input, der im data-Block formuliert wird (vgl. *Stan*-Code 3.1).

Stan-Code 3.1: Gemeinsamer data-Block der Modelle in *Stan*

```
,  
1 data {  
2     int<lower=0> T; // time points (equally spaced)  
3     int<lower=0, upper=1> miss_mask[T];  
4     vector[T] ret_obs; // Note: Masked indices will be treated as missing;  
5 }
```

3.1. Generalized Autoregressive Conditional Heteroscedasticity (GARCH)

Generalisierte autoregressive bedingte Heteroskedastizitäts-Modelle sind stochastische Modelle zur Zeitreihenanalyse.

Autoregressive Modelle beschreiben Beobachtungen als Kombination linearer Abhängigkeit von sich selbst sowie einer Zufallsvariable z. B. zufällige Modellfehler. Die Zufallsvariable ist entsprechend nur unter Unsicherheit vorhersagbar. Mit Hilfe von autoregressiven Modellen wird also untersucht inwiefern Beobachtungen in einem Zeitpunkt von der Vergangenheit abhängen.

Autoregressive bedingte Heteroskedastizitäts-Modelle kurz ARCH-Modelle wurden erstmals von Robert F. Engle im Jahr 1982 beschrieben. ARCH(i)-Modelle treffen die Annahme, dass die bedingte Varianz der zufälligen Modellfehler im Zeitpunkt t vom realisierten Zu-

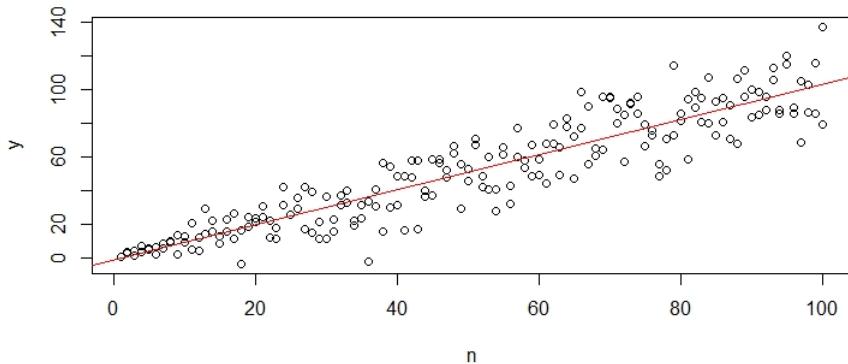


Abbildung 3.1.: Die Daten weisen im Zeitverlauf unterschiedliche Abweichung zur Trendgerade auf, d.h. die Residuen/Störterme sind nicht konstant im Zeitverlauf und es liegt Heteroskedastizität vor (Eigene Darstellung).

fallsfehler der letzten i Vorperioden abhängt, das heißt, dass große bzw. kleine Fehler dazu neigen in Clustern aufzutreten [7].

Heteroskedastizität ist ein Begriff der Statistik, der die Streuung innerhalb eines Datensatzes beschreibt. Heteroskedastizität liegt in einer Zeitreihe dann vor, wenn die Abweichung der Daten von einer Trendgeraden im Zeitverlauf zu- bzw. abnimmt oder auch einfach nicht in jedem Zeitpunkt gleich ist (vgl. Abbildung 3.1). Entsprechend bedeutet Homoskedastizität entsprechend, dass die Abweichung der Daten von der Trendgeraden in jedem Zeitpunkt gleich ist.

Eine Verallgemeinerung der ARCH-Modelle wurde 1986 von Tim Bollerslev in Form des GARCH(i,j)-Modells beschrieben. Hier hängt die bedingte Varianz der zufälligen Modellfehler im Zeitpunkt t nicht nur von Zufallsfehler der letzten i Vorperioden ab, sondern auch von sich selbst, also der bedingten Varianz der j vorherigen Zeitpunkte [5].

Sei d_t eine Zeitreihe und ϵ_t das Residuum im Zeitpunkt t , dann gilt

$$\begin{aligned} d_t &= \sigma_t \cdot \epsilon_t \\ \sigma_t^2 &= a_0 + a_1 d_{t-1}^2 + \dots + a_i d_{t-p}^2 + b_1 \sigma_{t-1}^2 + \dots + b_j \sigma_{t-j}^2 \end{aligned} \quad (3.1)$$

Wobei $a_l, b_k \in \mathbb{R}_{>0}$ mit $a_i \neq 0$ und $b_j \neq 0$ gilt. Außerdem sind die Fehlerterme unabhängige identisch verteilte Zufallsvariablen mit $E(\epsilon_t) = 0$ und $VAR(\epsilon_t) = 1$. Somit gilt, dass die bedingte Varianz

$$\sigma_t^2 = VAR(d_t | d_{t-1}, d_{t-2}, \dots)$$

sowohl von Werten der eigenen Vergangenheit als auch von der Vergangenheit der gesamten Zeitreihe abhängt.

In der Ökonometrie werden autoregressive Modelle z.B. für die Modellierung und Prognose von Finanzmarktdaten genutzt. Kurse an Finanzmärkten unterliegen in der Regel keiner gleichmäßigen Volatilität. Phasen mit geringen Schwankungen alternieren mit Phasen starker

Schwankungen. Die Zeitreihenanalyse und insbesondere GARCH-Modelle dienen der Identifikation und Beschreibung dieser Volatilitätscluster.

In dieser Arbeit wurde das GARCH(1,1)-Modell als Vergleichsmodell und Baseline berücksichtigt, da es ein in der Ökonometrie weitverbreitet genutztes Modell ist.

Beim GARCH(1,1)-Modell hängt die bedingte Varianz nur vom vorherigen Zeitpunkt ab und somit gilt

$$\sigma_t^2 = VAR(d_t|d_{t-1})$$

Entsprechend kann die GARCH(1,1) Zeitreihe wie folgt modelliert werden:

$$\begin{aligned} d_t &= \sigma_t \cdot \epsilon_t \\ \sigma_t^2 &= a_0 + a_1 d_{t-1}^2 + b_1 \sigma_{t-1}^2 \end{aligned} \quad (3.2)$$

Bei den untersuchten Zeitreihen in dieser Arbeit handelt es sich um Kursdaten des S&P 500 Indexes (vgl. Kapitel 4.1), d.h. das GARCH(1,1)-Modell untersucht Volatilitätscluster dieser Preise. Die Preise wurden in Erträge r_t umgerechnet, wobei je nach Markierung der Beobachtung der beobachtete Ertrag oder der geschätzte Ertrag r_t verwendet wird.

$$r_t = \mu + \sigma_t \cdot \epsilon_t$$

Entsprechend kann das GARCH(1,1)-Modell zur Modellierung der Volatilität in Form der Standardabweichung der Erträge implementiert werden.

$$\begin{aligned} \sigma_t^2 &= a_0 + a_1 d_{t-1}^2 + b_1 \sigma_{t-1}^2 \\ \sigma_t^2 &= a_0 + a_1 (\sigma_t \cdot \epsilon_t)^2 + b_1 \sigma_{t-1}^2 \\ \sigma_t^2 &= a_1 (\sigma_{t-1} \cdot \epsilon_{t-1} + \mu - \mu)^2 + b_1 \sigma_{t-1}^2 \\ \sigma_t^2 &= a_0 + a_1 (r_{t-1} - \mu)^2 + b_1 \sigma_{t-1}^2 \end{aligned} \quad (3.3)$$

Die Berechnung wird entsprechend im *Stan*-Code als transformierter Parameter implementiert.

```

1
2 parameters {
3   real mu;
4   real<lower=0> alpha0;
5   real<lower=0,upper=1> alpha1;
6   real<lower=0,upper=(1-alpha1)> beta1;
7   real<lower=0> sigma1;
8   real eps_miss[N]; // missing normalized return innovations
9 }
10 transformed parameters {
11   real ret[T]; // returns ... observed or r_t = mu + sigma_t * eps_t
12   real<lower=0> sigma[T];
13
14   {
15     int idx = 1; // missing value index

```

```

16
17 sigma[1] = sigmal;
18 if (miss_mask[1] == 1) {
19     ret[1] = mu + sigma[1] * eps_miss[idx];
20     idx = idx + 1;
21 } else
22     ret[1] = ret_obs[1];
23
24 for (t in 2:T) {
25     sigma[t] = sqrt(alpha0
26                     + alphal * pow(ret[t - 1] - mu, 2)
27                     + betal * pow(sigma[t - 1], 2));
28     if (miss_mask[t] == 1) {
29         ret[t] = mu + sigma[t] * eps_miss[idx];
30         idx = idx + 1;
31     } else
32         ret[t] = ret_obs[t];
33 }
34 }
35 }
```

Der Erwartungswert μ sowie die Standardabweichung σ sind standardnormalverteilt. Die Erträge folgen einer Normalverteilung mit den Parametern μ und σ . Außerdem muss die Ziel-Dichte die Jacobi-Korrektur (*engl. Jacobian Correction*) für alle Vorhersagen berücksichtigen. Schließlich wird im `model-block` des *Stan*-Codes, die a-posteriori-Dichte als normalisierte bedingte Verteilung der Erträge `ret_obs` von μ und σ definiert.

```

,
1 model {
2     mu ~ normal(0, 1);
3     sigmal ~ normal(0, 1);
4
5     ret ~ normal(mu, sigma);
6     // Jacobian correction for transformed innovations
7     for (t in 1:T) {
8         if (miss_mask[t] == 1)
9             target += log(sigma[t]);
10    }
11 }
12 generated quantities {
13     real log_lik[T];
14
15     for (t in 1:T)
16         log_lik[t] = normal_lpdf(ret_obs[t] | mu, sigma[t]);
17 }
```

Der vollständige *Stan*-Code des GARCH(1,1)-Modells befindet sich im Anhang A.1.

3.2. Franke & Westerhoff (FW)

R. Franke und F. Westerhoff haben ein Modell entwickelt, das den Markt durch zwei Agentengruppen bzw. Handelsstrategien beschreibt: Fundamental¹- und Chartist²-Trader. Händler können täglich die Strategie ändern, da im Modell ein Mechanismus modelliert wurde, der Herdenverhalten abbildet [9].

Der Anteil der Fundamental-Trader zum Zeitpunkt t sei

$$n_t^f \in [0, 1]$$

dann gilt für den Anteil der Chartist-Trader entsprechend

$$n_t^c = 1 - n_t^f$$

da nur zwei Handelsstrategien am Markt existieren[3, 9].

Der logarithmierte Preis p_t im Zeitpunkt t hängt vom vorherigen Preis sowie der vorherigen durchschnittlichen Nachfrage aller Händler d_{t-1}^f und d_{t-1}^c ab.

$$p_t = p_{t-1} + \mu(n_{t-1}^f \cdot d_{t-1}^f + n_{t-1}^c \cdot d_{t-1}^c) \quad (3.4)$$

Fundamental-Trader reagieren auf fehlerhafte Preise, z.B. wenn der Marktpreis vom (bekannten) Fundamentalpreis³ p^* des Vermögenswertes abweicht. Chartist-Trader reagieren im Gegensatz dazu auf Preisbewegungen der Vergangenheit z.B. $p_t - p_{t-1}$. Die Nachfrage selbst ist keine beobachtbare Größe und nur ihre gewichtete Summe beeinflusst die Preise. Sie wird von [3, 9] mittels folgender Dynamiken definiert:

$$\begin{aligned} d_t^f &= \phi(p^* - p_t) + \epsilon_t^f && \text{mit } \epsilon_t^f \sim \mathcal{N}(0, \sigma_f^2) \\ d_t^c &= \xi(p_t - p_{t-1}) + \epsilon_t^c && \text{mit } \epsilon_t^c \sim \mathcal{N}(0, \sigma_c^2) \\ &= \xi r_t + \epsilon_t^c \end{aligned}$$

Allgemein gilt für zwei unabhängige Zufallsvariablen X, Y , dass die Verteilung ihrer Summe der Faltung der Verteilung der beiden Zufallsvariablen entspricht.

Sei

$$\begin{aligned} X &\sim \mathcal{N}(\mu_X, \sigma_X^2) \\ Y &\sim \mathcal{N}(\mu_Y, \sigma_Y^2) \end{aligned}$$

¹Fundamental-Trader treffen Kauf bzw. Verkaufsentscheidung auf Basis vorhandener betriebswirtschaftlicher- und Kapitalmarktinformationen wie z.B. Jahresabschlüsse, Inflation, politische Informationen etc.. Für Fundamental-Trader gilt die Annahme, dass Märkte vorhersagbar sind und auf bestimmte Vorfälle vorhersagbares Verhalten zeigen. Durch Informationen über eben diese Vorfälle, kann der Händler informierte Vorhersagen über die Preise am Finanzmarkt treffen.

²Chartist-Trader treffen Handelsentscheidungen auf Basis historischer Preise und Volumen des Marktes. Chartist-Trader analysieren graphische Repräsentationen der historischen Daten und mathematische Indikatoren, um ihre Handelsstrategie zu optimieren.

³Der Fundamentalpreis ist der Preis, der rein auf Basis vorhandener Informationen berechnet wurde

dann gilt

$$X + Y \sim \mathcal{N}(\mu_X + \mu_Y, \sigma_X^2 + \sigma_Y^2)$$

So ist es möglich, dass die Nachfrage nicht tatsächlich berechnet bzw. geschätzt werden muss, sondern indirekt durch die Verteilung der kombinierten Nachfrage der existierenden Trader-Gruppen Einfluss auf die logarithmierten Preise $r_t = p_t - p_{t-1}$ nimmt. Für das stochastische Modell der logarithmierten Preise

$$r_t \sim \mathcal{N}(\mu_t, \sigma_t^2)$$

gelten dann die Verteilungsparameter

$$\begin{aligned}\mu_t &= E[n_{t-1}^f d_{t-1}^f + n_{t-1}^c d_{t-1}^c] \\ &= \mu(n_{t-1}^f \phi(p^* - p_{t-1}) + n_{t-1}^c \xi(p_{t-1} + p_{t-2}))\end{aligned}\quad (3.5)$$

und

$$\sigma_t^2 = \mu^2((n_{t-1}^f)^2 \sigma_f^2 + (n_{t-1}^c)^2 \sigma_c^2) \quad (3.6)$$

Die Volatilität σ_t hängt somit von der Größe der einzelnen Trader-Gruppen ab und ändert sich mit der Zeitreihe. Um den Wechsel der Händler von einer Gruppe zur anderen zu beschreiben, wird zu jedem Zeitpunkt t deren Anteil am Gesamtmarkt n_t^f bzw. n_t^c neu berechnet. Der Aktualisierungs-Quotient in Formel 3.7 entspricht der DCA⁴-HPM⁵ Spezifikation nach [3, 9].

Sei a_t die relative Attraktivität der Fundamental-Strategie gegenüber der Chartist-Strategie. Nach [3, 9] wird die Attraktivität mittels drei beeinflussender Faktoren modelliert (vgl. Formel 3.8). Zum einen wird eine allgemeine Veranlagung (General Predisposition) α_0 und zum anderen Herdenverhalten (Herding) α_n und fehlerhafte Preise α_p modelliert (HPM). Da nur eine diskrete Anzahl an Wahlmöglichkeiten bestehen (DCA), ist die Wahrscheinlichkeitsdichte des Modells leicht differenzierbar. Das hat zur Folge, dass ein HMC-Algorithmus leichter Stichproben aus der a-posterior-Verteilung ziehen kann.

Wie bereits erwähnt gilt

$$n_t^c = 1 - n_t^f$$

wobei die Veränderung der Anzahl der Händler zu jedem Zeitpunkt t durch den inversen Logit⁶ berechnet wird (vgl. Code 3.2 Zeile 53)

$$n_t^f = \frac{1}{1 + e^{-\beta \cdot a_t}} \quad \beta = 1 \text{ nach [3]} \quad (3.7)$$

mit

$$a_t = \alpha_0 + \alpha_n(n_t^f - n_t^c) + \alpha_p(p^* - p_t)^2 \quad a_n, a_p > 0 \quad (3.8)$$

Im Stan-Code werden die Parameter entsprechend im parameter- bzw. transformed parameter- Block definiert.

⁴Discrete Choice Approach(DCA) nach Formel 3.7 [3]

⁵Herding, Predisposition, Missalignment(HPM) Spezifikation für Attraktivität der Fundamental-Strategie gegenüber der Chartist-Strategie[3].

⁶Der Logit entspricht dem natürlichen Logarithmus der Wahrscheinlichkeit einer Chance (Quotient der Wahrscheinlichkeit p zur Gegenwahrscheinlichkeit $1 - p$). Für die Inverse gilt $p = \frac{e^x}{1+e^x} = \frac{1}{1+e^{-x}}$ [14]

Stan-Code 3.2: Teil-Modell nach [3, 9]

```

1  parameters {
2    real<lower=0> phi;
3    real<lower=0> xi;
4    real alpha_0;
5    real<lower=0> alpha_n;
6    real<lower=0> alpha_p;
7    real<lower=0> sigma_f;
8    real<lower=sigma_f> sigma_c;
9    real<lower=0, upper=1> n_f_1;
10   ...
11   vector[N] eps_miss; // missing normalized return innovations
12 }
13 transformed parameters {
14   vector[T] n_f;
15   vector[T] demand;
16   vector[T] sigma;
17   // Note: All prices are actually log prices!
18   vector[T] p_star;
19   vector[T] p;
20   real ret[T];
21   ...
22
23   p[1] = 0; // wlog log p_1 = 0
24   p_star[1] = ...
25
26   n_f[1] = n_f_1;
27   demand[1] = 0;
28
29   sigma[1] = mu * sqrt( square(n_f[1] * sigma_f)
30                         + square((1 - n_f[1]) * sigma_c));
31   {
32     int idx = 1;
33
34     if (miss_mask[1] == 1) {
35       ret[1] = sigma[1] * eps_miss[idx];
36       idx = idx + 1;
37     } else
38       ret[1] = ret_obs[1];
39
40     for (t in 2:T) {
41       // Note: index shift between prices and returns
42       p[t] = p[t - 1] + ret[t - 1];
43       p_star[t] = ...
44
45     {
46
47       // equation (HPM)
48       real a = alpha_n * (n_f[t-1] - (1 - n_f[t-1]))
49       + alpha_0
50       + alpha_p * square(p[t-1] - p_star[t-1]);

```

```

51
52     // equation (DCA)
53     n_f[t] = inv_logit(beta * a);
54
55     demand[t] = mu * (n_f[t] * phi * (p_star[t] - p[t])
56                         + (1 - n_f[t]) * xi * (p[t] - p[t-1]));
57
58     // structured stochastic volatility
59     sigma[t] = mu * sqrt( square(n_f[t] * sigma_f)
60                           + square((1 - n_f[t]) * sigma_c));
61   }
62
63   if (miss_mask[t] == 1) {
64     ret[t] = sigma[t] * eps_miss[idx];
65     idx = idx + 1;
66   } else
67     ret[t] = ret_obs[t];
68 }
69 }
70 }
```

Die Parameter β und μ sind redundante Parameter und werden konstant gehalten, da sie nur die Größenordnung der Parameter $\alpha_0, \alpha_n, \alpha_p, f, g, \sigma_f$ und σ_c beeinflussen.

Es gilt

$$\beta = 1$$

$$\mu = 0.01$$

Die Definition der Parameter wird im *Stan*-Code im `transformed data`-Block angelegt. Also enthält dieser neben der Berechnungsvorschrift der *Missing Values* auch die Vorschriften für σ_t , μ und β .

```

,
1 transformed data {
2   int N = 0; // number of missing values
3   real ret_sd = sqrt(variance(ret_obs));
4   // mu and beta fixed ... redundant anyways
5   real mu = 0.01;
6   real beta = 1.0;
7
8   for (t in 1:T)
9     if (miss_mask[t] == 1) N = N + 1;
10 }
```

Das Model wird anschließend im `model` bzw. `generated quantities`-Block mit finalisiert. Da wenig Wissen über die korrekte Wahl der a-priori-Verteilung für alle Parameter vorliegt, werden diese mit wenig informativen Verteilungen initialisiert, die die Skala der Parameter bestimmt. Dazu wurde die Studentsche t-Verteilung (vgl. Abbildung 3.2) gewählt, da diese schwere Ränder hat. So werden Werte, die um ein Vielfaches größer als die Standardabweichung sind, dennoch akzeptiert. Lediglich die Standardabweichungen folgen stärker informierte a-prori-Verteilungen auf Basis der Daten. Wie bereits in Kapitel 3.1 beschrieben wird auch hier die Jacobi-Korrektur und die a-posteriori-Verteilung modelliert.

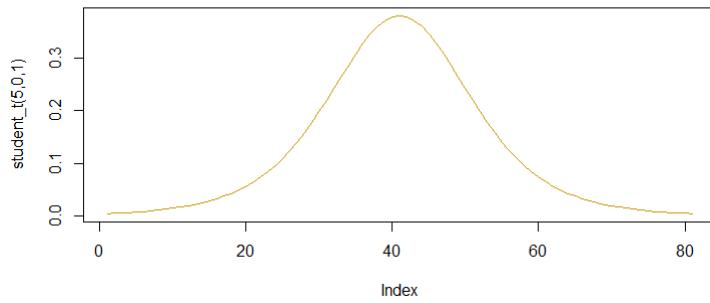


Abbildung 3.2.: Die Studentsche t-Verteilung mit 5 Freiheitsgraden, dem Erwartungswert 0 skaliert um den Faktor 1 (Eigene Darstellung).

```

1 model {
2   phi ~ student_t(5, 0, 1);
3   xi ~ student_t(5, 0, 1);
4   alpha_0 ~ student_t(5, 0, 1);
5   alpha_n ~ student_t(5, 0, 1);
6   alpha_p ~ student_t(5, 0, 1);
7   sigma_f ~ normal(0, ret_sd / mu);
8   sigma_c ~ normal(0, 2.0 * ret_sd / mu);
9   ...
10
11 // Price likelihood
12 ret ~ normal(demand, sigma);
13 // Jacobian correction for transformed innovations
14 for (t in 1:T) {
15   if (miss_mask[t] == 1)
16     target += log(sigma[t]);
17 }
18 }
19 generated quantities {
20   vector[T] log_lik;
21
22 for (t in 1:T)
23   log_lik[t] = normal_lpdf(ret_obs[t] | 0, sigma[t]);
24 }
```

Außerdem wird angenommen, dass der Fundamentalpreis unbekannt und nicht beobachtbar ist. Das wird durch den logarithmierte Fundamentalpreis $p^* = 0$ modelliert [3, 9]. Bei der Simulation des Modells auf realen Aktienmarktpreisen bzw. -erträgen, wird angenommen, dass sich der Fundamentalpreis im Zeitverlauf ändert.

Des Weiteren gilt, dass die Volatilität σ_t in Formel 3.6 über den Marktanteil der Fundamental-Trader n_{t-1}^f von der Attraktivität a_{t-2} in Formel 3.8 abhängt. Die Attraktivität wiederum wird

vom logarithmierten Fundamentalpreis p_{t-2}^* beeinflusst.

In der vorliegenden Arbeit wurden zwei verschiedene Varianten des Modells nach [3] auf Marktpreisen simuliert. Die Varianten unterscheiden sich in der Berechnung der Änderungen des logarithmierten Fundamentalpreises p_t^* :

1. Modellierung von p_t^* durch die Brownsche Bewegung bzw. eine Re-Parametrierung und die Berechnung mittels zufälliger Irrfahrt (Random Walk)
2. Modellierung von p_t^* durch einen gleitenden Durchschnitt (Moving Average)

In der ersten Variante wird angenommen, dass p_t^* im Zeitverlauf der geometrischen Brownschen Bewegung⁷ folgt.

$$p_t^* \sim \mathcal{N}(p_{t-1}^*, \sigma_*^2) \quad (3.9)$$

So enthält p_t^* durch die Brownsche Bewegung nun eine stochastische Komponente. Da p_t^* die Volatilität σ_t beeinflusst, handelt sich also um ein stochastisches Volatilitätsmodell [3, 9].

Durch eine nicht zentrierte Umparametrierung nach [3] kann die Performance des HCM-Sampler verbessert werden. Dazu wird p_t^* mittels ϵ_t^* als transformierter Parameter innerhalb einer zufälligen Irrfahrt (Random Walk) berechnet.

$$p_t^* = p_{t-1}^* + \sigma_* \cdot \epsilon_t^* \quad \text{wobei } \epsilon_t^* \sim \mathcal{N}(0, 1) \quad (3.10)$$

Das Modell bleibt dadurch unverändert, denn Formel 3.9 und 3.10 sind äquivalent [3].

Die erste Variante des Modells Θ_{FWwalk} wird also durch die Parameter ϵ_t^* und σ_* ergänzt. Im Stan-Code 3.3 ist diese Variante entsprechend im `parameters`- und `transformed parameters`-Block in Zeile 22 umgesetzt.

Stan-Code 3.3: Teil-Modell(Random Walk) nach [3, 9]

```

1 parameters {
2 ...
3 // p_star random walk in non-centered parameterization
4 vector[T] epsilon_star;
5 real<lower=0> sigma_p_star;
6 ...
7 }
8 transformed parameters {
9 ...
10 vector[T] p_star;
11 vector[T] p;
12 ...

```

⁷Die Brownsche Bewegung ermöglicht die Simulation multiplikativer unabhängiger Zuwächse des Fundamentalpreises [10]

```

13
14 p[1] = 0;
15 p_star[1] = p[1] + epsilon_star[1];
16
17 ...
18
19 for (t in 2:T) {
20     // Note: index shift between prices and returns
21     p[t] = p[t - 1] + ret[t - 1];
22     p_star[t] = p_star[t-1] + sigma_p_star * epsilon_star[t];
23
24 }
25 ...
26 }
27 model {
28 ...
29 epsilon_star ~ normal(0, 1);
30 sigma_p_star ~ normal(0, ret_sd / 2.0);
31 ...
32 }

```

Der vollständige *Stan*-Code der ersten Variante befindet sich im Anhang A.2.

In Variante 2 wird die Veränderung des logarithmierten Fundamentalpreises durch einen gleitenden Durchschnitt modelliert. Es wird angenommen, dass der Fundamentalpreis von aktuellen sowie vergangenen Werten eines unbekannten stochastischen Terms (Zufallsfehler) abhängt. Auch für diese Variante gilt somit, dass es sich um ein stochastisches Volatitätsmodell handelt [3].

Umgesetzt wird die Variante, indem p^* einer Mischverteilung folgt. Mischverteilungen repräsentieren die Wahrscheinlichkeitsverteilung der Beobachtungen einer Gesamtpopulation, die aus mehreren Sub-Populationen besteht. Der Vorteil der Mischverteilung ist, dass ein Datensatz einzelne Beobachtungen nicht tatsächlich einer Gruppe innerhalb der Gesamtpopulation zuordnen muss. Der Fundamentalpreis wird wie folgt modelliert:

$$p_t^* = P(p_t^* | \tau, p_{t-1}^*, p_t)$$

Also gilt

$$p_t^* \sim \text{mix}(\tau, p_{t-1}^*, p_t)$$

Dabei folgt der Skalierungsfaktor s einer inversen Gammaverteilung⁸.

$$s \sim \gamma^{-1}(2, 1)$$

$$\tau = e^{-\frac{1}{1000s}}$$

Im *Stan*-Code 3.4 werden die benötigten Parameter entsprechend definiert.

⁸In der Bayesschen Statistik wird die inverse Gammaverteilung häufig als marginale a-posteriori-Verteilung der unbekannten Varianz einer Normalverteilung genutzt [12].

Stan-Code 3.4: Teil-Modell(Moving Average) nach [3, 9]

```

1 parameters {
2 ...
3 real<lower=0> lenscale_raw;
4 real p_star_0;
5 ...
6 }
7 transformed parameters {
8 ...
9 vector[T] p_star;
10 vector[T] p;
11 ...
12 real<lower=0> lenscale = 1000 * lenscale_raw;
13 real<lower=0, upper=1> tau = exp( - 1 / lenscale);
14 ...
15 p[1] = 0; // wlog log p_1 = 0
16 p_star[1] = log_mix(tau, p_star_0, p[1]);
17 ...
18 for (t in 2:T) {
19     // Note: index shift between prices and returns
20     p[t] = p[t - 1] + ret[t - 1];
21     p_star[t] = log_mix(tau, p_star[t-1], p[t]);
22     ...
23 }
24 }
25 model {
26 ...
27 p_star_0 ~ normal(0, 0.2);
28 lenscale_raw ~ inv_gamma(2, 1); // avoid lower boundary ... lenscale ~ inv_gamma(2, 1000)
29 ...
30 }

```

Der vollständige Stan-Code befindet sich im Anhang A.3.

3.3. Alfarano, Lux & Wagner (AL)

Simulationen von Modellen, die komplexe Annahmen voraussetzen, weisen häufig stilisierte Fakten von Finanzmärkten, wie schwere Ränder der Verteilung der Erträge oder Volatilitätscluster auf. Allerdings können die Simulationen oft nicht mathematisch analysiert werden, da die Annahmen über die Marktdynamiken zu streng sind [1]. Das Modell nach Alfarano, Lux und Wagner [1] versucht Herdenverhalten in ein einfaches Gleichgewichtspreis-Modell zu integrieren, um strenge Annahmen bzgl. des Verhaltens der Händler oder der Handelsstrategie zu vermeiden und so beobachtete Muster auch für eine große Anzahl heterogener Agenten analytisch erklären zu können. Das Herdenverhalten wird in Form einer einfachen Variante des Herding-Modells nach Kirman [13] integriert.

Kirman hat 1993 auf Basis makroskopischer Muster aus entomologischen Experimenten mit

Ameisenkolonien ein Modell für Informationsaustausch stochastisch formalisiert [1]. Das Experiment beobachtet eine Ameisenkolonie, die in der Nähe ihres Nests zwei identische Nahrungsquellen zur Verfügung stehen. Es zeigt sich, dass einzelne Ameisen sich in der Regel auf eine Nahrungsquelle konzentrieren, aber dennoch gelegentlich zur anderen Nahrungsquelle wechseln. Betrachtet man den Mittelwert im Zeitverlauf ergibt sich eine bimodale⁹ Verteilung der Häufigkeit der Ameisen, die die eine bzw. andere Nahrungsquelle aufsuchen [1].

Alfarano, Lux und Wagner nutzen diesen Mechanismus, um Veränderungen im Verhalten der Händler zu formalisieren. Die binäre Auswahlmöglichkeit der Nahrungsquellen der Ameisen, gilt analog zur Entscheidung von Agenten des Finanzmarktes bzgl. einer bestimmten Regel zur Bildung ihrer Erwartung [1]. Auch in diesem Modell können Agenten am Markt zwischen Fundamental- und Chartist-Strategie entscheiden. Der Herding-Mechanismus bestimmt die Verteilung der Agenten auf diese beiden Handelsstrategien, sodass sich Phasen abwechseln in denen die Fundamental- bzw. Chartist-Strategie dominant ist. Wobei vor allem ein Markt mit vielen Chartist-Tradern zu Spekulationsblasen neigt [1].

Das Gleichgewicht des Marktes wird durch ein Standard Währungskurs-Modell bestimmt. Allerdings wurde die Annahme der rationalen Erwartungen durch nicht-rationale Erwartungen ersetzt. Diese nicht-rationalen Erwartungen entsprechen dem gewichteten Mittelwert der Erwartungen der zwei Händlergruppen, wobei die Gewichte einem Erfolgsmaß der Vergangenheit der jeweiligen Gruppe entsprechen. [1].

Am Markt existieren N Agenten, wobei n Agenten der ersten Handelsstrategie zugeordnet sind und entsprechend $N - n$ Agenten der anderen Strategie folgen. Im Zeitverlauf existieren zwei Möglichkeiten zwischen den zwei Strategien. Zum einen überzeugen zufällige Treffen manche Agenten die Strategie zu wechseln und zum anderen existieren autonome Wechsel, die z.B. durch unabhängige zufällige Zustandsänderungen modelliert werden können [1].

Im ursprünglichen Modell werden mehrfache Treffen von Agenten nicht berücksichtigt und die Wahrscheinlichkeiten einem anderen Agenten in eine andere Strategie zu folgen bzw. einen Agenten für eine Strategie zu gewinnen hängt nicht von der Vergangenheit ab. Mit Hilfe einer Markov-Kette können diese Abhängigkeiten modelliert werden, um Erinnerungen der Agenten zu simulieren [1]. Sei $x \in [-1, 1]$ die Konzentration oder Intensität der Agenten in der ersten Strategie mit

$$x = \frac{2n}{N - 1}$$

Die Wechsel bzw. Übergänge der Agenten von einer zur anderen Handelsstrategie sind wie folgt definiert:

$$\pi_x^\pm(x \rightarrow x \pm 2/N) = (1 \pm x)(2a/N + b(1 \pm x)) \quad (3.11)$$

Diese Übergänge hängen nicht vom relativen Anteil der Agenten einer Strategie bezogen auf die Gesamtpopulation ab, sondern kleinere Nachbarschaften/Subpopulationen beeinflussen die Entscheidung einzelner Agenten ihre Strategie zu wechseln. So werden sowohl zufällige

⁹Eine Wahrscheinlichkeitsverteilung heißt **bimodal**, wenn ihre Dichte zwei Modi aufweist.

Wechsel als auch Wechsel aufgrund von Herdenverhalten wie z.B. Gruppenzwang berücksichtigt, da z.B. eine große Anzahl aller Agenten der Gesamtpopulation der anderen Strategie folgen. Die Gleichgewichtsverteilung der Konzentration der Agenten $\omega(x)$ weist eine zentrale Tendenz auf, die z.B. durch den Mittelwert definiert sein kann. Somit können die Veränderungen des Systems vernachlässigt werden und die Dynamiken sind mittels deterministischer Differentialgleichung modellierbar.

So zeigen [1], dass x der Fokker-Planck-Gleichung¹⁰ mit Wahrscheinlichkeitsdichte $w_x(x, t)$ folgt.

$$\frac{\partial}{\partial t} w_x = -\frac{\partial}{\partial x} A(x) w_x + \frac{1}{2} \frac{\partial^2}{\partial x^2} D(x) w_x \quad (3.12)$$

mit Drift $A(x)$

$$\begin{aligned} A(x) &= \frac{N}{2} (\pi_x^+ - \pi_x^-) \\ &= -2ax \end{aligned} \quad (3.13)$$

und Diffusion $D(x)$

$$\begin{aligned} D(x) &= \pi_x^+ + \pi_x^- \\ &= 2b(1 - x^2) + \frac{4q}{N} \end{aligned} \quad (3.14)$$

Die Dynamiken werden also durch eine lineare Drift mit Mean-Reversion¹¹-Effekt Richtung des unbedingten Erwartungswerts $E[x] = 0$ modelliert. Außerdem besteht die Diffusion aus zwei Komponenten. Die erste Komponente ist eine symmetrische, quadratische Funktion von x , die unabhängig von der Gesamtpopulationsgröße N ist. Sie repräsentiert das Herdenverhalten aufgrund der Interaktion zwischen den Agenten. Die zweite Komponente hängt von N ab und repräsentiert die intrinsische Granularität des Modells. Sie wird für große N gegen 0 streben und ist somit zu vernachlässigen [1].

Befindet man sich nicht nah an den Rändern $x \pm 1$ dominiert der Herding-Mechanismus und die Veränderungen werden vom quadratischen Term repräsentiert, sodass der von N abhängige Term vernachlässigt werden kann. Befindet man sich allerdings an den Rändern, sind beide Effekte gleich groß und gleichbedeutend [1]. Es gilt, durch das kollektive Verhalten der Agenten und der nicht extensiv definierten Übergänge, dass die Größe der Schwankungen unabhängig von der Gesamtanzahl der Agenten N am Markt ist [1].

Die Fokker-Planck-Gleich kann ohne den von N abhängigen Term, in die Form einer klassischen stochastische Differentialgleichung dX_t umformuliert werden.

$$dX_t = \mu(X_t, t)dt + \sigma(X_t, t)dW_t$$

¹⁰Die **Fokker-Planck-Gleichung** beschreibt die zeitliche Entwicklung einer Wahrscheinlichkeitsdichte $P(x)$ unter der Wirkung von Drift $A(x, t)$ und Diffusion $B(x, t)$ [8].

¹¹Der **Mean-Reversion-Effekt** beschreibt die Annahme in der Kapitalmarkttheorie, dass Märkte zu Übertreibungen neigen, die sich im Zeitverlauf nicht zufällig korrigieren, sondern dass mittels Gedächtnis vorherige Trends umgekehrt werden. So folgt aus einem Anstieg der Preise fallende Kurse und umgekehrt [15].

Für die Fokker-Planck-Gleichung der Wahrscheinlichkeitsdichte $p(x, t)$ gilt dann

$$\frac{\partial}{\partial t} p(x, t) = -\frac{\partial}{\partial x} [\mu(x, t)p(x, t)] + \frac{1}{2} \frac{\partial^2}{\partial x^2} [D(x, t)p(x, t)] \quad (3.15)$$

wobei $D(x, t) = \sigma(x, t)$ gilt [2].

Daraus ergibt sich mit Drift und Diffusion aus Formel 3.13 und 3.14

$$dX_t = -2aX_t dt + \sqrt{2b(1 - X_t^2)} dW_t$$

Die Gleichung entspricht der Langevin-Gleichung nach [1], die mit Hilfe der Dynamiken der Konzentrationsvariable x die Preisdynamiken beschreibt. Die Langevin-Gleichung liefert eine Gaußsche Approximation an die stochastischen Dynamiken und das Modell kann nun mit Hilfe der Langevin-Gleichung simuliert werden. Betrachtet man die Dynamiken der relativen Preisänderungen

$$\begin{aligned} r(t, \Delta t) &= x(t + \Delta t) - x(t) \\ &= -2a\Delta t x(t) + \sqrt{2b\Delta t(1 - x^2)} \epsilon(t + \Delta t) \end{aligned}$$

wobei

$$x(t + \Delta t) = (1 - 2a\Delta t)x(t) + \sqrt{2b\Delta t(1 - x^2)} \epsilon(t) \quad (3.16)$$

wobei der Zufallsfehler ϵ einer Normalverteilung mit Mittelwert 0 folgt.

Somit gilt, dass die Konzentration x der Handelsstrategie im Zeitverlauf einer Normalverteilung folgt:

$$x \sim \mathcal{N}(\mu, \sigma^2)$$

Wobei

$$\mu = ((1 - 2a)x(t + \Delta t))$$

und

$$\sigma^2 = \sqrt{2b(1 - x(t + \Delta t)^2)}$$

Betrachtet man keine marginalen Zeitschritte, sondern, wie in dieser Arbeit, tägliche Änderungen der Preise und Zeitpunkte, erhält man die Beschreibung der Erträge:

$$\begin{aligned} r_t &= (x_{t+1} - x_t) + \sigma_f \epsilon \\ r(x, t) &\sim \mathcal{N} \left(\sum_{i=2}^{T+1} x_{t+1} - \sum_{i=1}^T x_t, \sigma_f \right) \end{aligned}$$

Wobei T der Anzahl der betrachteten Zeitpunkte entspricht. Außerdem gilt für $x \sim \mathcal{N}(\mu, \sigma^2)$

$$\mu = x_{t-1} - 2ax_{t-1}$$

und

$$\sigma^2 = \sqrt{2b(1 - x_{t-1}^2)}$$

Das ist dann auch entsprechend in *Stan*-Code 3.5 in Zeile 23 im `transformed parameter`-Block bzw. in Zeile 38 und 41 im `model`-Block umgesetzt.

Stan-Code 3.5: Teil-Modell nach [2, 1]

```

1  ,
2  transformed data {
3      int N = 0; // number of missing values
4      real T_quot = 1; // wlog fixed at 1
5
6      for (t in 1:T)
7          if (miss_mask[t] == 1) N = N + 1;
8  }
9  parameters {
10     vector<lower=0, upper=1>[T + 1] x; // sentiment over time
11     real<lower=0> sigma_f;
12     real<lower=0> alpha;
13     real<lower=0> beta;
14     vector[N] eps_miss; // missing normalized return innovations
15 }
16 transformed parameters {
17     vector[T] ret;
18
19     {
20         int idx = 1;
21
22         for (t in 1:T) {
23             if (miss_mask[t] == 1) {
24                 ret[t] = T_quot * (x[t + 1] - x[t]) + sigma_f * eps_miss[idx];
25                 idx = idx + 1;
26             } else
27                 ret[t] = ret_obs[t];
28         }
29     }
30 }
31 model {
32     sigma_f ~ normal(0, 1);
33     alpha ~ normal(0, 1);
34     beta ~ normal(0, 1);
35
36     // x[1] implicitly uniform
37     for (t in 2:(T+1))
38         x[t] ~ normal(x[t - 1] - 2.0 * alpha * x[t - 1],
39                         sqrt(2.0 * beta * (1 - square(x[t - 1]))));
40
41     // Price likelihood
42     ret ~ normal(T_quot * (x[2:(T+1)] - x[1:T]), sigma_f);
43     // Jacobian correction for transformed innovations
44     for (t in 1:T) {
45         if (miss_mask[t] == 1)
46             target += log(sigma_f);
47     }
48 }
```

Anschließend wird, wie bei den anderen Modellen im generated quantities-Block in Stan-Code 3.6, die a-posteriori-Verteilung in Zeile 6 und die Standardabweichung für jeden

Zeitpunkt σ_t in Zeile 7 modelliert.

Stan-Code 3.6: Teil-Modell nach [2, 1]

```
,  
1 generated quantities {  
2     vector[T] log_lik;  
3     vector[T] sigma;  
4  
5     for (t in 1:T) {  
6         log_lik[t] = normal_lpdf(ret_obs[t] | T_quot * (x[t + 1] - x[t]), sigma_f);  
7         sigma[t] = sqrt(square(T_quot * (x[t + 1] - x[t])) + square(sigma_f));  
8     }  
9 }
```

Der vollständige *Stan*-Code befindet sich im Anhang A.4.

4. Simulationen

Im folgenden Kapitel werden die Ergebnisse der Simulationen näher beschrieben. Kapitel 4.1 erläutert die verwendete Datenquelle bzw. die Zeitreihen und die Art der Vorhersagen. Im Kapitel 4.2 werden anschließend die Ergebnisse vorgestellt und mit Hilfe verschiedener Diagnoseplots diskutiert.

4.1. Daten & Vorhersagen

Um Dynamiken an Finanzmärkten zu erkennen, wurden zwei Zeitpunkte in der Vergangenheit gewählt, in denen starke Dynamiken zu beobachten waren. Zum einen die globale Finanzkrise mit Höhepunkt im September 2008 und zum anderen die Dotcom-Blase mit Höhepunkt im März 2000.

Alle vier Modelle wurden auf beiden Zeitreihen simuliert, indem jeder Monat des Jahres 2000 bzw. 2008 auf Basis der Daten der vorherigen acht Jahre simuliert wurde. Als Daten wurden tägliche Preise des Aktienindexes *S&P 500*¹ genutzt, so dass ein Monat annähernd durch 30 Vorhersage-Zeitpunkte abgebildet werden konnte.

Die Datensätze von YAHOO! FINANCE enthalten jeweils sowohl den Eröffnungs- als auch den Schlusskurs pro Tag. Daraus wurden die logarithmierten Erträge berechnet, die als Input für die Modelle genutzt werden können. In R kann dazu mit Hilfe des Pakets tidyverse der Dataframe *data* um eine berechnete Spalte erweitert werden: `data <- data %>% mutate(LogRet = log(Cl - Wo / log hier für den natürlichen Logarithmus steht.`

In der Finanzmathematik ist es üblich mit logarithmierten Erträgen zu arbeiten, insbesondere wenn die Berechnung von Volatilitäten im Vordergrund steht. Logarithmierte Erträge sind stetige Erträge und auf der gesamten Menge der reellen Zahlen definiert. Diskrete Erträge sind häufig nur auf den positiven Zahlen definiert bzw. auf 100% Verlust begrenzt. Durch die Verwendung stetiger Erträge kann die empirische Verteilung der Renditen besser durch stochastische Verteilungen approximiert werden.

Mit Hilfe der a-posteriori-Verteilung werden dann logarithmierte Erträge für 30 Tage berechnet. Diese Vorhersagen wurden für alle vier Chains in einem Plot visualisiert. Dazu wurden die logarithmierten Erträge wieder in Preise umgerechnet und anschließend die Konfidenzintervalle 0.95, 0.8 und 0.5 des Medians der Preise als Streuungsbreite eines Linienplots der tatsächlichen Preise gezeichnet. Die exakte Berechnung des Plots befindet sich im *R-Skript 1* in Anhang B.

¹Datenquelle: YAHOO! FINANCE <https://finance.yahoo.com/quote/%5EGSPC>

4.2. Ergebnisse

Um die Modelle möglichst gut auf den Daten fitten zu können, können dem Sampler/der Markov-Kette bestimmte Parameter übergeben werden. In dieser Arbeit wurden dazu teilweise der `max_treedepth`- sowie der `adapt_delta`-Parameter verwendet. Die Parameter wirken zwei aufgetretenen Problemen entgegen:

Zum einen, dass die Berechnungen einer Iteration des Samplers die max. Tiefe des Baums, der in einer Iteration evaluiert werden soll, überschritten hat. Dabei handelt es sich um ein Effizienzproblem, dass durch den `max_treedepth` Parameter die Tiefe der vom Sampler aufgebauten Bäume kontrolliert.

Zum anderen, dass *divergent Transitions* auftreten. Die Iterationen der Markov-Kette, die an den geometrisch schwierigen Bereichen hängen bleiben oder diese nicht erreichen können und schließlich vom Sampler abgelehnt werden, bezeichnet man als divergent Transitions. Der Parameter `adapt_delta` legt die Schrittweite fest, um so auch aus schwierigen Bereichen der Ziel-Verteilung Stichproben zu generieren ohne divergent Transitions zu verursachen. Je näher der Wert an 1 gewählt wird, desto kleiner werden die Schrittweiten gewählt. Das führt wiederum auch dazu, dass die Bäume, die in einer Iteration evaluiert werden müssen, tiefer werden und eventuell muss auch der Parameter `max_treedepth` angepasst werden.

Als Fehlermeldung zum Modell werden dann zum Beispiel die Anzahl der divergent Transitions bzw. die Anzahl der Iterationen, welche die `max_treedepth` überschritten haben zurückgemeldet. Es wird allerdings nur festgehalten welche der Iterationen eine divergent Transition war und es ist nicht klar nachvollziehbar in welchem Bereich der Verteilung oder zu welchem Zeitpunkt diese Transition tatsächlich abgelehnt wurde. Die Markov-Kette kann sich durch das Monte-Carlo-Sampling allerdings schnell und weit fortbewegen bzw. große Schritte Sprünge machen, sodass nicht eindeutig festgestellt werden kann, wo genau in der Verteilung die divergent Transitions auftreten.

Eine allgemeine Übersicht der Ergebnisse der einzelnen Modelle auf den verschiedenen Daten und welche Parameter jeweils verwendet wurden befindet sich in Anhang C in Abbildung C.1.1 bis C.1.3.

Der Abbildung ist zu entnehmen, dass das FW-Modell, insbesondere die Variante mit dem gleitenden Durchschnitt (`FW_DCA_HPM_ma_CV`), auf den Daten der Dotcom-Blase extrem schwer zu simulieren war. Auf den Daten der Finanzkrise hatte der Sampler mit beiden FW-Modell Varianten bei entsprechend hohem `adapt_delta` Wert auf den ersten Blick weniger Probleme die a-posteriori-Verteilung zu erkennen. Auf den Daten der Dotcom-Blase war es für beide Varianten (`FW_DCA_HPM_walk_CV`) schon deutlich schwerer ohne eine hohe Anzahl divergenter Transitions das Sampling zu beenden. Die Erhöhung des `adapt_delta` über 0.95 führte dazu, dass das Sampling überhaupt nicht beendet werden konnte bzw. so langsam wurde, dass auch mehrere Tage Laufzeit nicht ausreichend für den Sampler waren, d.h. auch mit sehr kleinen Schrittweiten konnte die Markov-Kette die a-posteriori-Verteilung nicht berechnen (vgl. Abbildung 4.1).

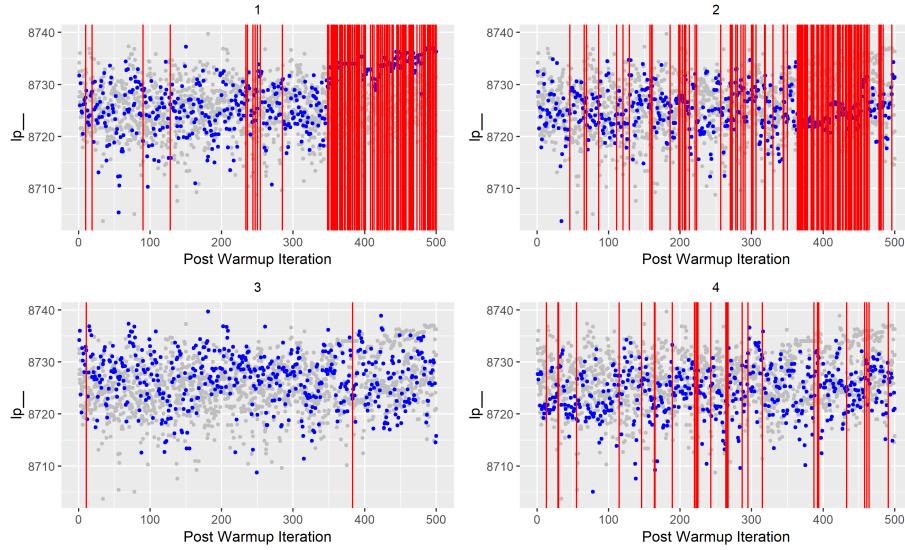


Abbildung 4.1.: Traceplots der Chains des FW-Modells(ma) auf den Daten der Dotcom-Blase für August 2008 mit 381 divergent Transitions und 8 Transitions über `max_treedepth = 18` wobei `adapt_delta = 0.9` (eigene Darstellung)

Auf den Daten der Finanzkrise hatte das FW-Modell ebenfalls Probleme. Oft hatte eine der Chains eine deutlich geringere Wahrscheinlichkeit als die anderen oder die Schätzungen der einzelnen Chains liegen für einzelne Parameter sehr weit auseinander, obwohl alle Chains ähnliche Wahrscheinlichkeit haben (vgl. Abbildung 4.2 sowie Abbildung 4.3 und 4.4).

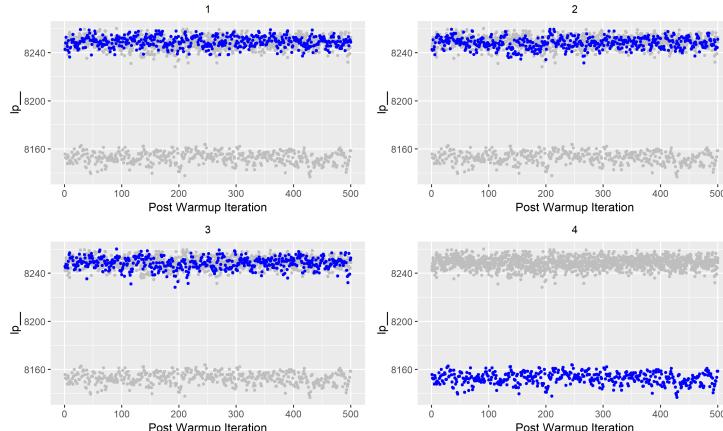


Abbildung 4.2.: Traceplots der Chains des FW-Modells (ma) auf den Daten der Finanzkrise für Juli 2008 mit 0 divergent Transitions und 0 Transitions über `max_treedepth = 18` wobei `adapt_delta = 0.99` (eigene Darstellung)

Auch das FW-Modell mit der Irrfahrt zeigt nur sehr breit gestreute Vorhersagen und die stärker gebündelten Vorhersagen einzelner Chains zeigen eine geringere Wahrscheinlichkeit

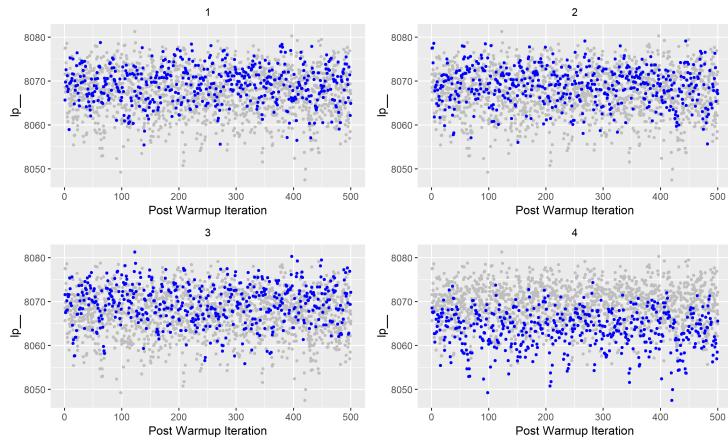


Abbildung 4.3.: Traceplots der Chains des FW-Modells (ma) auf den Daten der Finanzkrise für November 2008 mit 0 divergent Transitions und 0 Transitions über `max_treedepth = 29` wobei `adapt_delta = 0.99` (eigene Darstellung)

(vgl. Abbildung 4.5 und 4.6).

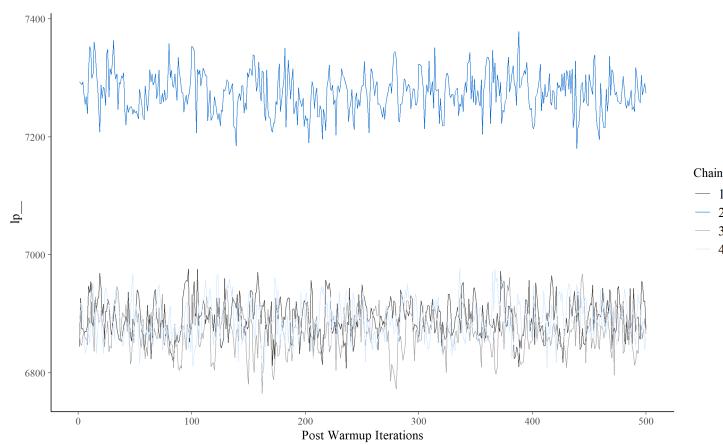


Abbildung 4.5.: Traceplot der Chains des FW-Modells (walk) auf den Daten der Finanzkrise für Dezember 2008 mit 0 divergent Transitions und 0 Transitions über `max_treedepth = 128` wobei `adapt_delta = 0.99` (eigene Darstellung)

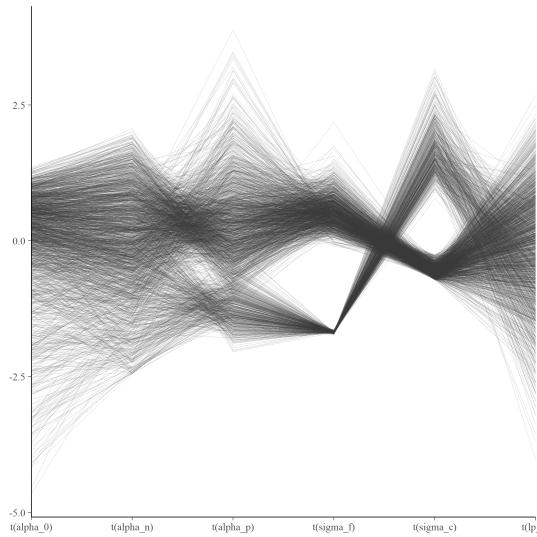


Abbildung 4.4.: Parallele-Koordinaten-Plot der Chains des FW-Modells (ma) auf den Daten der Finanzkrise für November 2008 mit 0 divergent Transitions und 0 Transitions über `max_treedepth = 29` wobei `adapt_delta = 0.99` (eigene Darstellung)

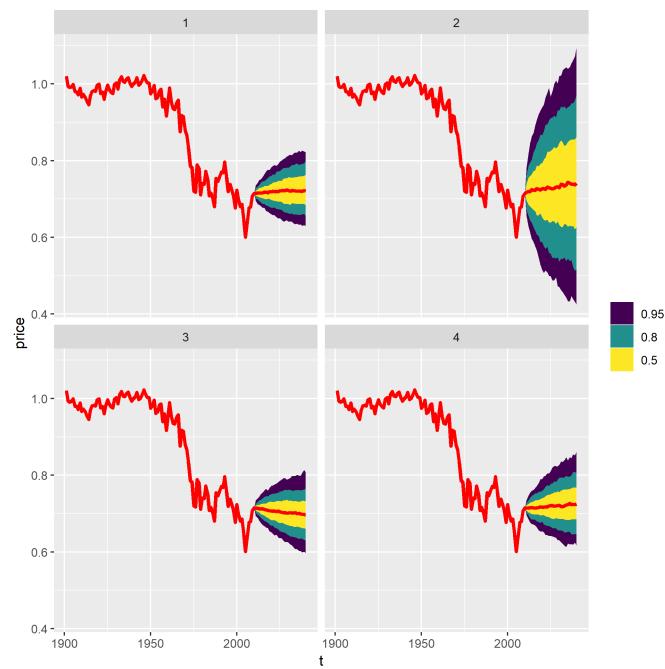


Abbildung 4.6.: Preis-Vorhersage der Chains des FW-Modells (walk) auf den Daten der Finanzkrise für Dezember 2008 mit 0 divergent Transitions und 0 Transitions über `max_treedepth = 128` wobei `adapt_delta = 0.99` (eigene Darstellung)

Die Ergebnisse lassen vermuten, dass das FW-Modell die a-posteriori-Verteilung der Finanzmarkt-Daten in extremen Krisen nicht ausreichend erfassen kann. Eine Änderung des Modells könnte die Performance verbessern.

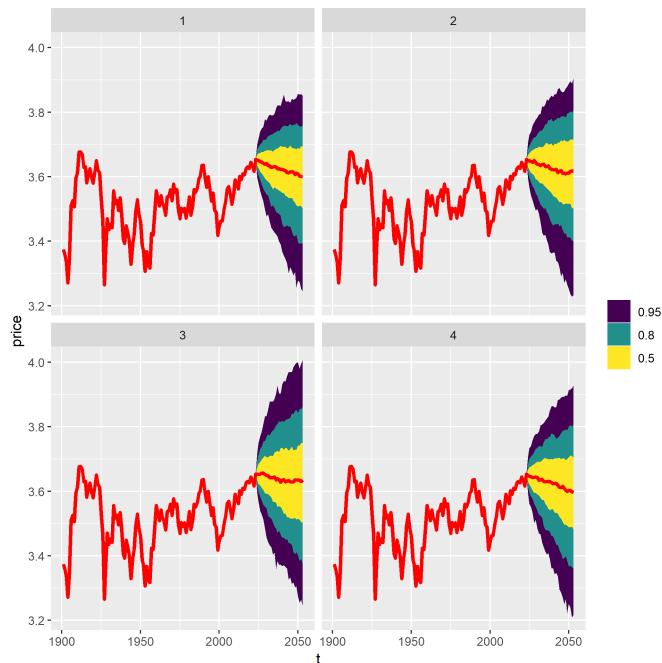


Abbildung 4.7.: Preis-Vorhersage der Chains des AL-Modells auf den Daten der Dotcom-Blase für September 2000 mit 2 divergent Transitions und 0 Transitions über `max_treedepth = 16` wobei `adapt_delta = 0.92` (eigene Darstellung)

Das AL-Modell zeigt z.B. im September 2000 eher eine Tendenz zu fallenden Preisen (vgl. Abbildung 4.7). Alle Chains zeigen eine ähnlich hohe Wahrscheinlichkeit. Nur Chain 3 zeigt zu Beginn starke Abweichungen, die sich im Verlauf des Samplings, dann allerdings wieder stark an die Werte der anderen Chains annähern. Die Dynamik ist der Abbildung 4.8 zu entnehmen. Das erklärt mitunter die breitere Streuung der Vorhersagen von Chain 3 im Vergleich zu den anderen Chains in Abbildung 4.7.

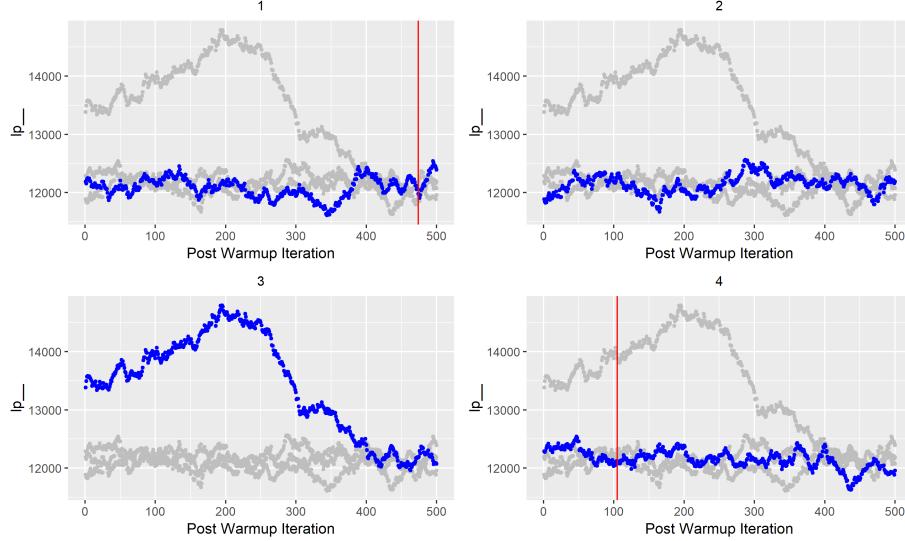


Abbildung 4.8.: Traceplots der Chains des AL-Modells auf den Daten der Dotcom-Blase für September 2000 mit 2 divergent Transitions und 0 Transitions über `max_treedepth = 16` wobei `adapt_delta = 0.92` (eigene Darstellung)

Es scheint auf den ersten Blick, dass das AL-Modell leichter und mit weniger Problemen auf beiden Datensätzen zu fitten ist. Die genauere Betrachtung offenbart allerdings, dass auch dieses Modell nur breit gestreute Vorhersagen liefert. Außerdem zeigt insbesondere der Parameter β konstant hohe Autokorrelation auf beiden Datensätzen, was bedeutet, dass sich die Chains zwischen Iterationen nicht viel bewegt haben sondern im gleichen Bereich geblieben sind. Im besten Fall strebt die Autokorrelation sehr schnell gegen 0 oder hat negative Werte, was bedeutet, dass sich der Durchschnittswert des Parameters der Stichprobe seinem wahren Wert annähert (vgl. Abbildung 4.9 und 4.10). Eventuell kann das Ergebnis durch eine Erhöhung der Anzahl der Iterationen oder eine Änderung des Parameters β verbessert werden.

Das GARCH-Modell läuft ebenfalls auf beiden Datensätzen ohne Fehlermeldung bei Verwendung der Standardwerte für die Parameter `max_treedepth` und `adapt_delta`. Die Vorhersagen sind breit gestreut und lassen teilweise auf den Daten der Dotcom-Blase mit sehr geringer Wahrscheinlichkeit eine Tendenz zu steigenden Preisen vermuten (vgl. Abbildung 4.11). Grundsätzlich sind die Vorhersagen aber sehr breit gestreut und es sind keine Dynamiken erkennbar.

Die Autokorrelation der Parameter strebt für alle Monate und jede Zeitreihe, genau wie für beide Varianten des FW-Modells, schnell gegen 0 und ist teilweise negativ. (vgl. Abbildung ??) Die Wahrscheinlichkeiten der Chains liegen immer sehr nah beieinander und es gibt keine Ausreißer. Das liegt daran, dass das GARCH-Modell wenige Parameter nutzt und diese nicht von den Daten selbst abhängen. Das führt auch dazu, dass die Ergebnisse auf beiden Zeitreihen keine Unterschiede zeigen.

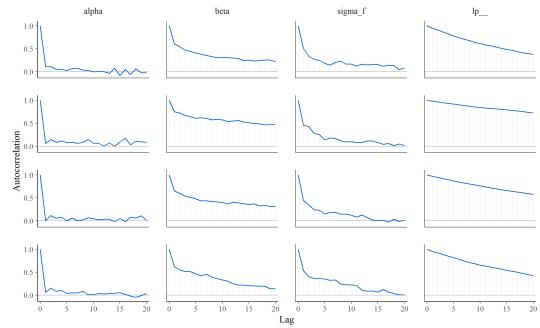


Abbildung 4.9.: Autokorrelation der Modell-Parameter innerhalb der Chains des AL-Modells auf den Daten der Dotcom-Blase für Juni 2000 mit 1 divergent Transitions und 0 Transitions über `max_treedepth = 16` wobei `adapt_delta = 0.9` (eigene Darstellung)

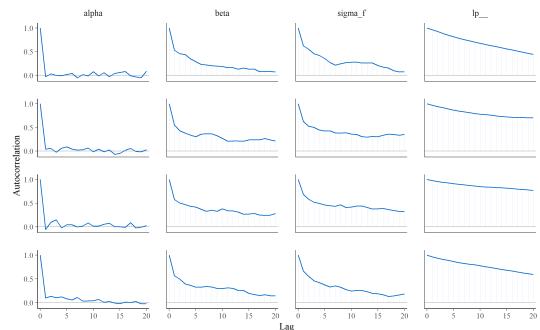


Abbildung 4.10.: Autokorrelation der Modell-Parameter innerhalb der Chains des AL-Modells auf den Daten der Finanzkrise für Dezember 2008 mit 4 divergent Transitions und 0 Transitions über `max_treedepth = 13` wobei `adapt_delta = 0.85` (eigene Darstellung)

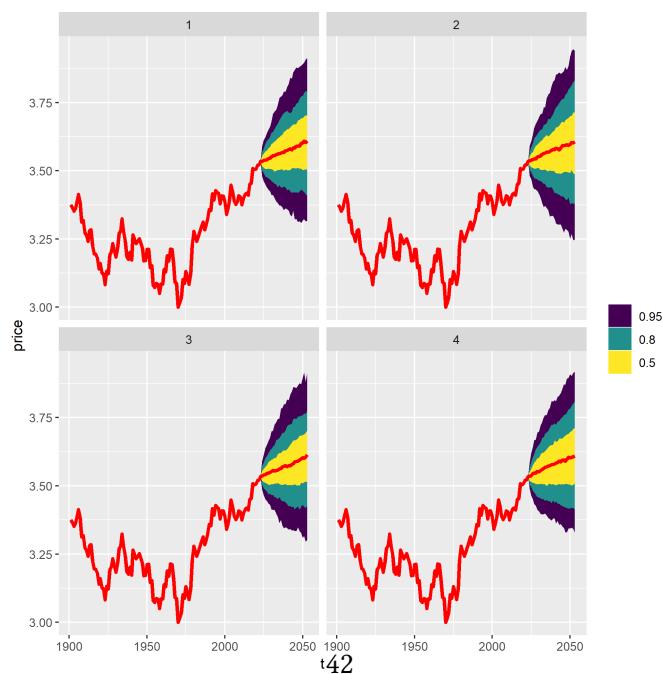


Abbildung 4.11.: Preis-Vorhersage der Chains des GARCH-Modells auf den Daten der Dotcom-Blase für Januar 2000 mit 0 divergent Transitions und 0 Transitions über `max_treedepth = 10` wobei `adapt_delta = 0.8` (eigene Darstellung)

Im Allgemeinen konnte keins der vier Modelle realistische Dynamiken erkennen. Die Vorhersagen mit hohen Wahrscheinlichkeiten waren weitestgehend breit gestreut und es war nicht klar erkennbar, ob die Preise eher steigen oder fallen würden. Vorhersagen mit geringer Sicherheit konnten teilweise Dynamiken zeigen.

Anhang

A. Stan Code

A.1. GARCH-Modell

```
,  
1 data {  
2     int<lower=0> T;  
3     int<lower=0, upper=1> miss_mask[T];  
4     real ret_obs[T]; // Note: Masked indices will be treated as missing;  
5 }  
6 transformed data {  
7     int N = 0; // number of missing values  
8     for (t in 1:T)  
9         if (miss_mask[t] == 1) N = N + 1;  
10 }  
11 parameters {  
12     real mu;  
13     real<lower=0> alpha0;  
14     real<lower=0,upper=1> alphal;  
15     real<lower=0,upper=(1-alphal)> betal;  
16     real<lower=0> sigmal;  
17     real eps_miss[N]; // missing normalized return innovations  
18 }  
19 transformed parameters {  
20     real ret[T]; // returns ... observed or r_t = mu + sigma_t * eps_t  
21     real<lower=0> sigma[T];  
22  
23     {  
24         int idx = 1; // missing value index  
25  
26         sigma[1] = sigmal;  
27         if (miss_mask[1] == 1) {  
28             ret[1] = mu + sigma[1] * eps_miss[idx];  
29             idx = idx + 1;  
30         } else  
31             ret[1] = ret_obs[1];  
32  
33         for (t in 2:T) {  
34             sigma[t] = sqrt(alpha0  
35                             + alphal * pow(ret[t - 1] - mu, 2)  
36                             + betal * pow(sigma[t - 1], 2));  
37             if (miss_mask[t] == 1) {  
38                 ret[t] = mu + sigma[t] * eps_miss[idx];  
39                 idx = idx + 1;  
40             } else
```

```

41         ret[t] = ret_obs[t];
42     }
43 }
44 }
45 model {
46     mu ~ normal(0, 1);
47     sigma ~ normal(0, 1);
48
49     ret ~ normal(mu, sigma);
50     // Jacobian correction for transformed innovations
51     for (t in 1:T) {
52         if (miss_mask[t] == 1)
53             target += log(sigma[t]);
54     }
55 }
56 generated quantities {
57     real log_lik[T];
58
59     for (t in 1:T)
60         log_lik[t] = normal_lpd\phi (ret_obs[t] | mu, sigma[t]);
61 }
```

A.2. Modell von Franke & Westerhoff (Random Walk)

```

,
1 data {
2     int<lower=0> T; // time points (equally spaced)
3     int<lower=0, upper=1> miss_mask[T];
4     vector[T] ret_obs; // Note: Masked indices will be treated as missing;
5 }
6 transformed data {
7     int N = 0; // number of missing values
8     real ret_sd = sqrt(variance(ret_obs));
9     // mu and beta fixed ... redundant anyways
10    real mu = 0.01;
11    real beta = 1.0;
12
13    for (t in 1:T)
14        if (miss_mask[t] == 1) N = N + 1;
15 }
16 parameters {
17     real<lower=0> phi;
18     real<lower=0> xi;
19     real alpha_0;
20     real<lower=0> alpha_n;
21     real<lower=0> alpha_p;
22     real<lower=0> sigma_f;
23     real<lower=sigma_f> sigma_c;
24     real<lower=0, upper=1> n_f_1;
25     // p_star random walk in non-centered parameterization
26     vector[T] epsilon_star;
```

```

27  real<lower=0> sigma_p_star;
28  vector[N] eps_miss; // missing normalized return innovations
29 }
30 transformed parameters {
31  vector[T] n_f;
32  vector[T] demand;
33  vector[T] sigma;
34  // Note: All prices are actually log prices!
35  vector[T] p_star;
36  vector[T] p;
37  real ret[T];
38
39  p[1] = 0; // wlog log p_1 = 0
40  p_star[1] = p[1] + epsilon_star[1]; // FIXME ... interpretation epsilon_raw[1]
41
42  n_f[1] = n_f_1;
43  demand[1] = 0;
44  sigma[1] = mu * sqrt( square(n_f[1] * sigma_f)
45    + square((1 - n_f[1]) * sigma_c));
46 {
47  int idx = 1;
48
49  if (miss_mask[1] == 1) {
50    ret[1] = sigma[1] * eps_miss[idx];
51    idx = idx + 1;
52  } else
53    ret[1] = ret_obs[1];
54
55  for (t in 2:T) {
56    // Note: index shift between prices and returns
57    p[t] = p[t - 1] + ret[t - 1];
58    p_star[t] = p_star[t-1] + sigma_p_star * epsilon_star[t];
59
60    {
61    // equation (HPM)
62    real a = alpha_n * (n_f[t-1] - (1 - n_f[t-1]))
63      + alpha_0
64      + alpha_p * square(p[t-1] - p_star[t-1]);
65    // equation (DCA)
66    n_f[t] = inv_logit(beta * a);
67    demand[t] = mu * (n_f[t] * phi * (p_star[t] - p[t])
68      + (1 - n_f[t]) * xi * (p[t] - p[t-1]));
69    // structured stochastic volatility
70    sigma[t] = mu * sqrt( square(n_f[t] * sigma_f)
71      + square((1 - n_f[t]) * sigma_c));
72  }
73
74  if (miss_mask[t] == 1) {
75    ret[t] = sigma[t] * eps_miss[idx];
76    idx = idx + 1;
77  } else
78    ret[t] = ret_obs[t];

```

```

79     }
80   }
81 }
82 model {
83   phi ~ student_t(5, 0, 1);
84   xi ~ student_t(5, 0, 1);
85   alpha_0 ~ student_t(5, 0, 1);
86   alpha_n ~ student_t(5, 0, 1);
87   alpha_p ~ student_t(5, 0, 1);
88   sigma_f ~ normal(0, ret_sd / mu);
89   sigma_c ~ normal(0, 2.0 * ret_sd / mu);
90   epsilon_star ~ normal(0, 1);
91   sigma_p_star ~ normal(0, ret_sd / 2.0);

92
93   // Price likelihood
94   ret ~ normal(demand, sigma);
95   // Jacobian correction for transformed innovations
96   for (t in 1:T) {
97     if (miss_mask[t] == 1)
98       target += log(sigma[t]);
99   }
100 }
101 generated quantities {
102   vector[T] log_li;
103
104   for (t in 1:T)
105     log_li[t] = normal_lpd\phi (ret_obs[t] | 0, sigma[t]);
106 }
```

A.3. Modell von Franke & Westerhoff (Moving Average)

```

,
1 data {
2   int<lower=0> T; // time points (equally spaced)
3   int<lower=0, upper=1> miss_mask[T];
4   vector[T] ret_obs; // Note: Masked indices will be treated as missing;
5 }
6 transformed data {
7   int N = 0; // number of missing values
8   real ret_sd = sqrt(variance(ret_obs));
9   // mu and beta fixed ... redundant anyways
10  real mu = 0.01;
11  real beta = 1.0;
12
13  for (t in 1:T)
14    if (miss_mask[t] == 1) N = N + 1;
15 }
16 parameters {
17   real<lower=0> phi;
18   real<lower=0> xi;
19   real alpha_0;
```

```

20  real<lower=0> alpha_n;
21  real<lower=0> alpha_p;
22  real<lower=0> sigma_f;
23  real<lower=0> sigma_c;
24  real<lower=0, upper=1> n_f_1;
25  real<lower=0> lenscale_raw;
26  real p_star_0;
27  vector[N] eps_miss; // missing normalized return innovations
28 }
29 transformed parameters {
30  vector[T] n_f;
31  vector[T] demand;
32  vector[T] sigma;
33  // Note: All prices are actually log prices!
34  vector[T] p_star;
35  vector[T] p;
36  real ret[T];
37  real<lower=0> lenscale = 1000 * lenscale_raw;
38  real<lower=0, upper=1> tau = exp(-1 / lenscale);
39
40  p[1] = 0; // wlog log p_1 = 0
41  p_star[1] = log_mix(tau, p_star_0, p[1]);
42
43  n_f[1] = n_f_1;
44  demand[1] = 0;
45  sigma[1] = mu * sqrt(square(n_f[1] * sigma_f)
46    + square((1 - n_f[1]) * sigma_c));
47  {
48    int idx = 1;
49
50    if (miss_mask[1] == 1) {
51      ret[1] = sigma[1] * eps_miss[idx];
52      idx = idx + 1;
53    } else
54      ret[1] = ret_obs[1];
55
56    for (t in 2:T) {
57      // Note: index shift between prices and returns
58      p[t] = p[t - 1] + ret[t - 1];
59      p_star[t] = log_mix(tau, p_star[t-1], p[t]);
60
61      {
62        // equation (HPM)
63        real a = alpha_n * (n_f[t-1] - (1 - n_f[t-1]))
64        + alpha_0
65        + alpha_p * square(p[t-1] - p_star[t-1]);
66        // equation (DCA)
67        n_f[t] = inv_logit(beta * a);
68        demand[t] = mu * (n_f[t] * phi * (p_star[t] - p[t])
69          + (1 - n_f[t]) * xi * (p[t] - p[t-1]));
70        // structured stochastic volatility
71        sigma[t] = mu * sqrt(square(n_f[t] * sigma_f)

```

```

72     + square((1 - n_f[t]) * sigma_c));
73 }
74
75 if (miss_mask[t] == 1) {
76     ret[t] = sigma[t] * eps_miss[idx];
77     idx = idx + 1;
78 } else
79     ret[t] = ret_obs[t];
80 }
81 }
82 }
83 model {
84     phi ~ student_t(5, 0, 1);
85     xi ~ student_t(5, 0, 1);
86     alpha_0 ~ student_t(5, 0, 1);
87     alpha_n ~ student_t(5, 0, 1);
88     alpha_p ~ student_t(5, 0, 1);
89     sigma_f ~ normal(0, ret_sd / mu);
90     sigma_c ~ normal(0, 2.0 * ret_sd / mu);
91     p_star_0 ~ normal(0, 0.2);
92     lenscale_raw ~ inv_gamma(2, 1); // avoid lower boundary ... lenscale ~ inv_gamma(2, 1000)
93
94 // Price likelihood
95 ret ~ normal(demand, sigma);
96 // Jacobian correction for transformed innovations
97 for (t in 1:T) {
98     if (miss_mask[t] == 1)
99         target += log(sigma[t]);
100 }
101 }
102 generated quantities {
103     vector[T] log_liks;
104
105     for (t in 1:T)
106         log_liks[t] = normal_lpdf(ret_obs[t] | 0, sigma[t]);
107 }
```

A.4. Modell von Alfarano, Lux & Wagner (AL)

```

,
1 data {
2     int<lower=0> T; // time points (equally spaced)
3     int<lower=0, upper=1> miss_mask[T];
4     vector[T] ret_obs; // Note: Masked indices will be treated as missing;
5 }
6 transformed data {
7     int N = 0; // number of missing values
8     real T_quot = 1; // wlog fixed at 1
9
10    for (t in 1:T)
11        if (miss_mask[t] == 1) N = N + 1;
```

```

12 }
13 parameters {
14   vector<lower=0, upper=1>[T + 1] x; // sentiment over time
15   real<lower=0> sigma_f;
16   real<lower=0> alpha;
17   real<lower=0> beta;
18   vector[N] eps_miss; // missing normalized return innovations
19 }
20 transformed parameters {
21   vector[T] ret;
22
23 {
24   int idx = 1;
25
26   for (t in 1:T) {
27     if (miss_mask[t] == 1) {
28       ret[t] = T_quot * (x[t + 1] - x[t]) + sigma_f * eps_miss[idx];
29       idx = idx + 1;
30     } else
31       ret[t] = ret_obs[t];
32   }
33 }
34 }
35 model {
36   sigma_f ~ normal(0, 1);
37   alpha ~ normal(0, 1);
38   beta ~ normal(0, 1);
39
40   // x[1] implicitly uniform
41   for (t in 2:(T+1))
42     x[t] ~ normal(x[t - 1] - 2.0 * alpha * x[t - 1],
43                   sqrt(2.0 * beta * (1 - square(x[t - 1]))));
44
45   // Price likelihood
46   ret ~ normal(T_quot * (x[2:(T+1)] - x[1:T]), sigma_f);
47   // Jacobian correction for transformed innovations
48   for (t in 1:T) {
49     if (miss_mask[t] == 1)
50       target += log(sigma_f);
51   }
52 }
53 generated quantities {
54   vector[T] log_lik;
55   vector[T] sigma;
56
57   for (t in 1:T) {
58     log_lik[t] = normal_lpdf(ret_obs[t] | T_quot * (x[t + 1] - x[t]), sigma_f);
59     sigma[t] = sqrt(square(T_quot * (x[t + 1] - x[t])) + square(sigma_f));
60   }
61 }
```

B. R Code

B.1. Erstellen der Preis-Plots

Stan-Code 1: R-Script zur Visualisierung der Preis-Vorhersagen

```
,  
1  
2 #libraries  
3 library(rstan)  
4 library(tidybayes)  
5 library(tidyverse)  
6  
7 stanfitmodels <- c(AL_herd_Jun, AL_herd_Jul)  
8  
9  
10 priceplot(stanfitmodels, 6, "dotcom", "png", "plots/")  
11  
12 priceplot <- function(fits, start_month, data, save_as, path){  
13   m <- start_month  
14  
15   for (fit in fits){  
16  
17     month <- month.abb[m]  
18  
19     #get spread draws of fit in tidy structure  
20     fit_draws <- spread_draws(fit, ret[t]) %>%  
21       group_by(.chain, .iteration) %>%  
22       arrange(t) %>%  
23       #calculate prices from log return  
24       mutate(price = exp(cumsum(ret[t]))) %>%  
25       group_by(t,.chain) %>%  
26       #get intervals  
27       median_qi(price, .width = c(.5,.8,.95))  
28  
29     #plot lineribbon of prices  
30  
31     assign(paste0(month, "_price_prediction"), fit_draws %>% filter(t > 1900) %>% ggplot(aes(x  
= t, y = price)) +  
32       geom_lineribbon() +  
33       facet_wrap(nrow = 2,~.chain), envir = .GlobalEnv)  
34  
35     # save plot  
36     ggsave(paste0(month,"_price_prediction_", fit@model_name,"_",".", data, ".", save_as),device =  
37     save_as, path = path)  
38  
39     #next month  
40     m <- m + 1  
41   }  
41 }
```

B.2. Import/Export der Stanfit Objekte

Stan-Code 2: R-Script zum Speichern und Importieren der Stanfit-Objekte aus bzw. in den aktuellen Workspace

```
,  
1 library(tidyverse)  
2  
3 stanfitmodels <- c(AL_herd_Jan, AL_herd_Feb, AL_herd_Mar, AL_herd_Apr, AL_herd_May, AL_herd_Jun  
 , AL_herd_Jul)  
4  
5 import <- c("AL_herd_Jan", "AL_herd_Feb")  
6  
7  
8 save_fits(stanfitmodels, 1, "dotcom", "stanfit/")  
9  
10 load_fits("AL_herd_Mar", 1,"AL_herd_walk_CV", "dotcom", "dotcom/AL_herd/stanfit/")  
11  
12  
13  
14 save_fits <- function(fits, start_month, data, path){  
15   m <- start_month  
16  
17   for (fit in fits){  
18     month <- month.abb[m]  
19     write_rds(fit, paste0(path, month, "_fit_", data, "_", fit@model_name , ".rds"))  
20     m <- m + 1  
21   }  
22 }  
23  
24 load_fits <- function(fits, start_month,model_name, data, path){  
25   m <- start_month  
26  
27   for (fit in fits){  
28     month <- month.abb[m]  
29     assign(paste0(month, "_AL_"), data), read_rds(paste0(path, month, "_fit_", data, "_", model  
 _name , ".rds")), envir = .GlobalEnv)  
30     m <- m + 1  
31   }  
32 }
```

B.3. Erstellen der Traceplots inkl. divergent Transitions für einzelne Chains

```
,  
1  
2 # load libraries  
3 library(rstan)  
4 library(tidyverse)  
5 library(tidybayes)  
6 library(bayesplot)  
7 library(grid)
```

```

8 library(gridExtra)
9
10 #example call
11 custom_trace_each_chain_overlay(AL_herd_Feb, "Feb", "dotcom", 4, "png", "plots/")
12
13 #definition
14 custom_trace_each_chain_overlay <- function(fit, month,data, chains, save_as, path){
15
16   #get div trans for each iteration
17   divtrans <- get_divergent_iterations(fit)
18   #get lp_ for each iteration
19   lp <- as.data.frame(fit)$lp_
20
21   #set chain for each iteration (sequential)
22   chain <- c()
23   t <- c()
24   #set counter for 1 to 500 post warm up iterations for each chain
25   c <- 1
26   for (i in 1:length(lp)){
27     if(i<= 500){
28       chain <- c(chain, 1)
29     }
30     if(i>500 & i <= 1000){
31       chain <- c(chain, 2)
32     }
33     if(i>1000 & i <= 1500){
34       chain <- c(chain, 3)
35     }
36     if(i>1500){
37       chain <- c(chain, 4)
38     }
39     if (c == 501){
40       c <- 1
41     }
42     t <- c(t, c)
43     c <- c + 1
44   }
45
46   #convert to data frame
47   divergence <- as.data.frame(t)
48   divergence[".chain"] <- chain
49   divergence["lp_"] <- lp
50   divergence["div"] <- as.numeric(divtrans)
51
52   plots <- list()
53   for (i in 1:chains){
54     plots[[i]] <- ggplot() +
55       geom_point(data = divergence %>% filter(.chain != i), aes(x=t, y = lp_), color = "gray",
56       size = 1) +
57       geom_point(data = divergence %>% filter(.chain == i), aes(x=t, y = lp_), color = "blue",
58       size = 1) +
59       geom_vline(data = divergence %>% filter(div == 1, .chain == i) ,aes(xintercept= t), color

```

```

      = "red") +
  ylim(min(divergence$lp_),max(divergence$lp_)) +
  xlab("Post Warmup Iteration") +
  ggtitle(i) +
  theme(plot.title = element_text(hjust = 0.5, size = 10))
}

#generate plot grid
plot_grid <- grid.arrange(grobs = plots, nrow = 2)
#save plot
ggsave(paste0(month, "_trace_all_chains_custom_overlay_", fit@model_name,"_", data, ".", save_as), plot = arrangeGrob(grobs = plots, ncol = 2), device = save_as , path = path)

return(plot_grid)
}

```

B.4. Erstellen der Diagnose-Plots für mehrere Stanfit-Objekte

Stan-Code 3: R-Script zur Erstellung verschiedener Diagnose-Plots

```

,
1 # load libraries
2 library(rstan)
3 library(tidyverse)
4 library(tidybayes)
5 library(bayesplot)
6 library(grid)
7 library(gridExtra)
8

10 #function to standardize over values of parameter of whole model
11 standard <- function(x) {(x - mean(x)) / sd(x)}
12

13 #specify parameter of model to be used in plots
14 param <- c( "sigma_f","alpha", "beta", "lp_")
15

16 #specifiy stanfit models to be plotted
17 stanfitmodels <- c(AL_herd_Jan, AL_herd_Feb, AL_herd_Mar, AL_herd_Apr )
18

19 #example call - generate all plots for each fit
20 generate_plots(stanfitmodels, 1, param, "mix-brightblue-gray", "dotcom", "png", "plots/")
21

22 #example call - trace of single chain per plot (line chart)
23 custom_trace_each_chain(AL_herd_Apr, "Apr", "pdf")
24 #example call - one highlighted chain per plot (point chart)
25 custom_trace_each_chain_overlay(AL_herd_Feb, "Feb", "dotcom", 4, "png", "plots/")
26

27

28 #definition:
29 generate_plots <- function(fits, start_month, parameter, color_scheme,data, save_as, path){

```

```

30
31 #set color scheme
32 color_scheme_set(color_scheme)
33
34 #set lp var as last elem of input parameter vector par
35 lp <- tail(parameter, n=1)
36
37 #generate plots for each fit/month
38 month <- c()
39 m <- start_month
40 for (fit in fits){
41   #generate month names to be used for plot objects and store month abbreviation in a vector
42
43   month <- month.abb[m]
44   # START: standard trace plot
45   assign(paste0(month, "_trace"), mcmc_trace(as.array(fit), pars = lp, np = nuts_params(fit))
46   ) + xlab("Post Warmup Iterations"), envir = .GlobalEnv)
47   #save plot
48   ggsave(paste0(month, "_traces_"), fit@model_name,"_", data, ".", save_as), device =
49   save_as , path = path)
50
51   # END: standard trace plot
52
53   # START: custom trace plot for each chain
54   #assign(paste0(month[m], "_trace_chains_custom"), custom_trace_each_chain(fit, month[m],
55   data, save_as, path), envir = .GlobalEnv)
56
57   # no need to save - handled by function which creates plot
58
59   # custom trace plot: chain overlay
60   assign(paste0(month, "_trace_all_chains_overlay"), custom_trace_each_chain_overlay(fit,
61   month, data , 4, save_as, path), envir = .GlobalEnv)
62
63   # END: custom trace plot for each chain
64
65   # START: pairs plot
66
67   assign(paste0(month, "_pairs"), mcmc_pairs(as.array(fit), pars = parameter, np =
68   nuts_params(fit)), envir = .GlobalEnv)
69   #save plot
70   dev.copy2pdf(file = paste0(path, month, "_pairs_"), fit@model_name,"_", data, ".pdf"))
71   dev.off()
72
73   # ggsave not working-- prints only last plot
74   #ggsave(paste0(month, "_pairs_"), fit@model_name,"_", data, ".", save_as), device = save_as
75   , path = path)
76   # END: pairs plot
77
78   # START: parcoord plot
79   assign(paste0(month, "_parcoord"), mcmc_parcoord(as.array(fit), pars = parameter, np =
80   nuts_params(fit),transform = standard), envir = .GlobalEnv)

```

```

75  #save plot
76  ggsave(paste0(month, "_parcoord_"), fit@model_name,"_", data, ".", save_as), device =
77  save_as , path = path)
# END: parcoord plot
78
79  # START: visualize posterior with and without divergent transitions
80  assign(paste0(month, "_div"), mcmc_nuts_divergence(nuts_params(fit),log_posterior(fit)),
81  envir = .GlobalEnv)
#save plot
82  ggsave(paste0(month, "_div_"), fit@model_name,"_", data, ".", save_as), device = save_as ,
83  path = path)
84  for (j in 1:4) {
85    assign(paste0(month, "_div_chain"), j), mcmc_nuts_divergence(nuts_params(fit),
86    log_posterior(fit),chain=j), envir = .GlobalEnv)
#save plot
87  ggsave(paste0(month, "_div_chain", j, "_"), fit@model_name,"_", data, ".", save_as),
88  device = save_as , path = path)
}
89
90  # START: autocorrelation
91  assign(paste0(month, "_autocorrelaton"), mcmc_acf(as.array(fit),pars = parameter), envir =
92  .GlobalEnv)
#save plot
93  ggsave(paste0(month, "_autocorrelation_"), fit@model_name,"_", data, ".", save_as), device =
94  save_as , path = path)
# END: autocorrelation
95
96  # START: energy level / bfmi low error - distribution on energy level during mcmc sampling
97  assign(paste0(month, "_energy"), mcmc_nuts_energy(nuts_params(fit),binwidth = 5), envir =
98  .GlobalEnv)
#save plot
99  ggsave(paste0(month, "_energy_"), fit@model_name,"_", data, ".", save_as), device = save_as ,
100  path = path)
# END_ energy level
101
102  #next month
103  print(paste(month, "done"))
104  m <- m + 1
105 }
106
107 }
108
109 #custom traceplot if each chain has 500 post warmup iterations
110 custom_trace_each_chain <- function(fit, month,data, save_as, path){
111
112  #get div trans for each iteration
113  divtrans <- get_divergent_iterations(fit)
114  #get lp__ for each iteration
115  lp <- as.data.frame(fit)$lp__
116
117  #set chain for each iteration (sequential)

```

```

118 chain <- c()
119 t <- c()
120 #set counter for 1 to 500 post warm up iterations for each chain
121 c <- 1
122 for (i in 1:length(lp)){
123   if(i<= 500){
124     chain <- c(chain, 1)
125   }
126   if(i>500 & i <= 1000){
127     chain <- c(chain, 2)
128   }
129   if(i>1000 & i <= 1500){
130     chain <- c(chain, 3)
131   }
132   if(i>1500){
133     chain <- c(chain, 4)
134   }
135   if (c == 501){
136     c <- 1
137   }
138   t <- c(t, c)
139   c <- c + 1
140 }
141
142 #convert to data frame
143 divergence <- as.data.frame(t)
144 divergence[".chain"] <- chain
145 divergence["lp_"] <- lp
146 divergence["div"] <- as.numeric(divtrans)
147
148 #draw line plot + vertical line for each divergent transition
149 plot <- ggplot() +
150   geom_line(data = divergence, aes(x=t, y = lp_), color = "blue") +
151   geom_vline(data = divergence %>% filter(div == 1) ,aes(xintercept= t), color = "red") +
152   ylim(min(divergence$lp_),max(divergence$lp_)) +
153   xlab("Post Warmup Iteration") +
154   facet_wrap(~.chain)
155
156 #save plot
157 ggsave(paste0(month, "_trace_all_chains_custom_", fit@model_name,"_", data, ".", save_as),
158         device = save_as , path = path)
159
160 return(plot)
161 }
162
163 custom_trace_each_chain_overlay <- function(fit, month,data, chains, save_as, path){
164
165   #get div trans for each iteration
166   divtrans <- get_divergent_iterations(fit)
167   #get lp_ for each iteration
168   lp <- as.data.frame(fit)$lp_

```

```

169 #set chain for each iteration (sequential)
170 chain <- c()
171 t <- c()
172 #set counter for 1 to 500 post warm up iterations for each chain
173 c <- 1
174 for (i in 1:length(lp)){
175   if(i<= 500){
176     chain <- c(chain, 1)
177   }
178   if(i>500 & i <= 1000){
179     chain <- c(chain, 2)
180   }
181   if(i>1000 & i <= 1500){
182     chain <- c(chain, 3)
183   }
184   if(i>1500){
185     chain <- c(chain, 4)
186   }
187   if (c == 501){
188     c <- 1
189   }
190   t <- c(t, c)
191   c <- c + 1
192 }
193
194 #convert to data frame
195 divergence <- as.data.frame(t)
196 divergence[".chain"] <- chain
197 divergence["lp__"] <- lp
198 divergence["div"] <- as.numeric(divtrans)
199
200 plots <- list()
201 for (i in 1:chains){
202   plots[[i]] <- ggplot() +
203     geom_point(data = divergence %>% filter(.chain != i), aes(x=t, y = lp__), color = "gray",
204     size = 1) +
205     geom_point(data = divergence %>% filter(.chain == i), aes(x=t, y = lp__), color = "blue",
206     size = 1) +
207     geom_vline(data = divergence %>% filter(div == 1, .chain == i) ,aes(xintercept= t), color =
208     "red") +
209     ylim(min(divergence$lp__),max(divergence$lp__)) +
210     xlab("Post Warmup Iteration") +
211     ggtitle(i) +
212     theme(plot.title = element_text(hjust = 0.5, size = 10))
213 }
214
215
216 #generate plot grid
217 plot_grid <- grid.arrange(grobs = plots, nrow = 2)
218 #save plot
219 ggsave(paste0(month, "_trace_all_chains_custom_overlay_", fit@model_name, "_", data, ".",
220           save_as), plot = arrangeGrob(grobs = plots, ncol = 2), device = save_as , path = path)

```

```

217
218
219     return(plot_grid)
220 }

```

B.5. Ausführung der Stan-Modelle in R

Stan-Code 4: R-Script zum Import von Daten aus einer CSV-Datei und Erstellen der Stanfit Objekte

```

,
1 library(tidyverse)
2
3 #import data from CSV
4 data <- read_csv("../data/dot-com/Dec_1992-2000_SP500_USD.csv",
5                   col_types = cols(
6                     Date = col_date(),
7                     Open = col_double(),
8                     High = col_double(),
9                     Low = col_double(),
10                    Close = col_double(),
11                    'Adj Close' = col_double(),
12                    Volume = col_number()
13                  ))
14
15 #calculate returns
16 data <- data %>% mutate(Ret = Close - Open)
17 data <- data %>% mutate(LogRet = log(Close)-log(Open))
18
19 #mark imported data as not missing
20 data <- data %>% mutate(Missing = 0)
21
22
23 # add predictions
24 date <- as.Date(tail(data$Date, 1))
25 for (x in 0:29) {
26   date <- date + 1
27   data <- add_row(data, Date = date , Ret = 0,LogRet = 0, Missing = 1)
28 }
29
30 # Load Rstan package
31 library(rstan)
32 rstan_options(auto_write = TRUE)
33 options(mc.cores = parallel::detectCores())
34
35 #set parameter and save stanfit model as 'AL_herd_Dec'
36 AL_herd_Dec <- stan(control = list(max_treedepth = 16, adapt_delta=.99),
37                       file="../models/AL_herd_walk_CV.stan",
38                       data = list(T=nrow(data), miss_mask = data$Missing,
39                                 ret_obs = data$LogRet),
40                       iter = 1000,
41                       chains = 4)

```

B.6. Erstellen der Allgemeinen Plots zur Übersicht der Parameter der Modelle

Stan-Code 5: R-Script zur Erstellung von Übersichten der Parameter innerhalb einer Zeitreihe

```
,  
1 #load libraries  
2 library(tidyverse)  
3 library(rstan)  
4  
5 #define data and exakt model names that match .rds file names  
6 data <- c("dotcom", "finance")  
7 model_name <- c("AL_herd_walk_CV", "FW_DCA_HPM_ma_CV", "FW_DCA_HPM_walk_CV", "GARCH_CV")  
8  
9 #define function that returns a list of all .rds files (file structur sensitiv)  
10 generate_import_incl_path <- function (data, model_name){  
11   fit_vec <- c()  
12   data_vec <- c()  
13   model_vec <- c()  
14  
15   for (model in model_name){  
16     for (d in data){  
17       path <- paste0(d, "/", model, "/stanfit/")  
18       for (i in 1:12){  
19         fit <- c(paste0(path, month.abb[i], "_fit_", d, "_", model))  
20         fit_vec <- c(fit_vec, fit)  
21         data_vec <- c(data_vec, d)  
22         model_vec <- c(model_vec, model)  
23       }  
24     }  
25   }  
26   return(list(fit_vec, data_vec, model_vec))  
27 }  
28  
29  
30 #define function that loads all .rds files and extracts data, returns data frame  
31 run <- function(fits, data, model){  
32  
33   month <- c()  
34   month_abb <- c()  
35   num_div <- c()  
36   num_tree <- c()  
37   adapt_delta <- c()  
38   max_treedepth <- c()  
39   time <- c()  
40   d <- c()  
41   m <- c()  
42  
43   i <- 1  
44   j <- 1  
45  
46   for (elem in fits){  
47     #load to workspace  
48     fit <- read_rds(paste0(elem, ".rds"))
```

```

49
50 #add month
51 month <- c(month, i)
52 month_abb <- c(month_abb, month.abb[i])
53
54 #add data
55 d <- c(d, data[j])
56
57 #add model
58 m <- c(m, model[j])
59
60 # get model spec
61 #add number divergent transitions
62 num_div <- c(num_div, get_num_divergent(fit))
63
64 #add number transitions exceeding max treedepth
65 num_tree <- c(num_tree, get_num_max_treedepth(fit))
66
67 #add elapsed time
68 time <- c(time, mean(get_elapsed_time(fit)))
69
70 #get delta
71 adapt_delta <- c(adapt_delta, attr(fit@sim$samples[[1]], "args")$control[[1]])
72
73 #get max treedepth
74 max_treedepth <- c(max_treedepth, attr(fit@sim$samples[[1]], "args")$control[[9]])
75
76 if(i == 12){
77   i <- 1
78 } else {
79   i <- i + 1
80 }
81 # counter to access data and model name from fits
82 j <- j + 1
83 }
84 #convert to dataframe
85 general <- as.data.frame(month)
86 general['month_abb'] <- month_abb
87 general['num_divergence'] <- num_div
88 general['exceed_max_treedepth'] <- num_tree
89 general['elapsed_time'] <- time
90 general['adapt_delta'] <- adapt_delta
91 general['max_treedepth'] <- max_treedepth
92 general['data'] <- d
93 general['model'] <- m
94
95 return(general)
96 }
97
98 # generate import list
99 import <- generate_import_incl_path(data, model_name)
100

```

```

101 #save data frame on disk
102 write_rds( run(import[[1]], import[[2]], import[[3]]), paste0("general_all_models.rds"))
103
104 #load data frame
105 general_all_models <- read_rds("general_all_models.rds")
106
107 # accessing data from data frame "general_all_models to check specification from model run
108 general_all_models %>% filter(data == data[1]) %>% filter(model == model_name[1]) %>% filter(
109   month == 9)
110
111 # example plot of general information
112
113 general_all_models %>% ggplot() +
114   geom_line(aes(x=month, y = num_divergence)) +
115   geom_text(aes(label = num_divergence,x = month, y = num_divergence), size = 3, vjust = -1) +
116   scale_x_continuous(name="month", breaks = 1:12) +
117   facet_wrap(model ~ data, ncol = 2)

```

C. Weitere Abbildungen

C.1. Allgemeine Plots

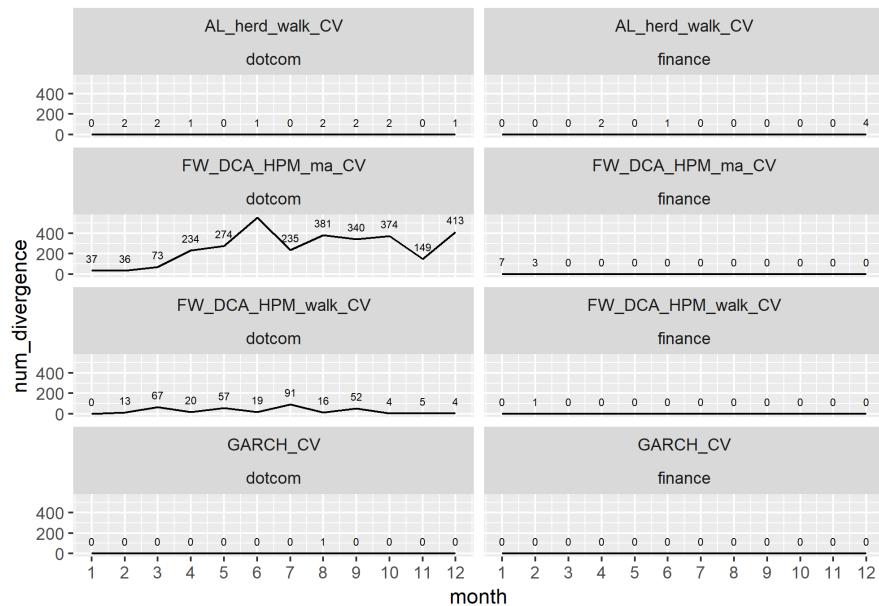


Abbildung C.1.1.: Übersicht über die Anzahl der divergenten Transitions je Monat aller Modelle auf den verschiedenen Daten (eigene Darstellung)

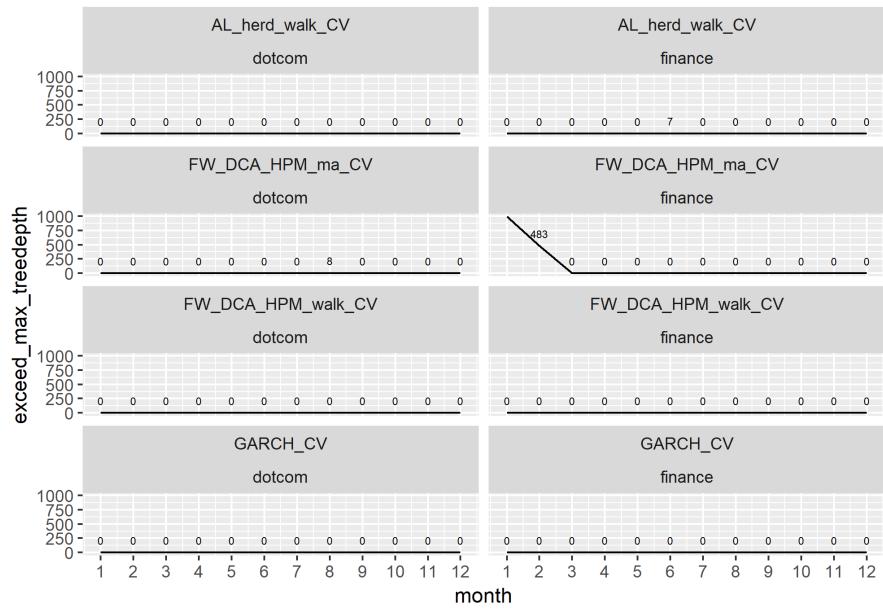


Abbildung C.1.2.: Übersicht über die Anzahl der Iterationen, die die `max_treedepth` überschritten haben je Monat, Modell und Datensatz (eigene Darstellung)

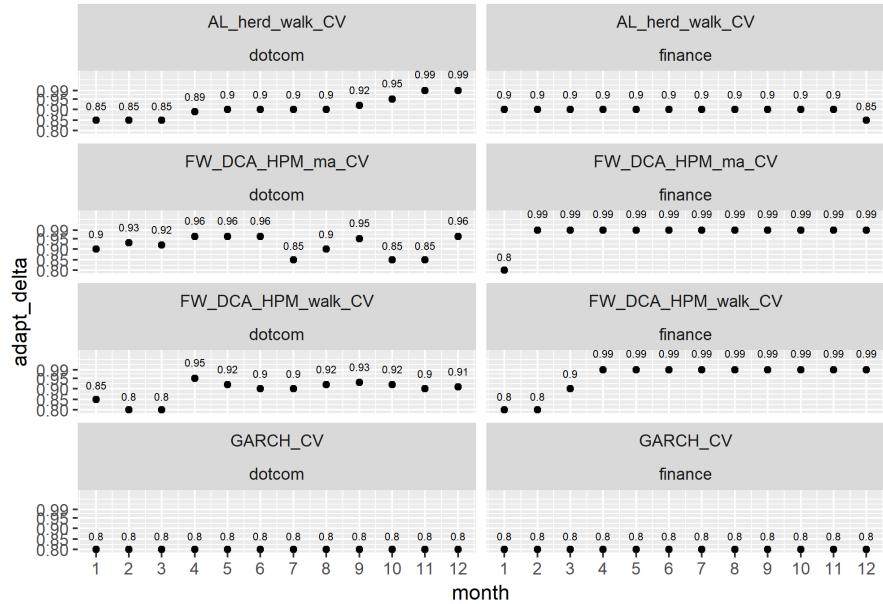


Abbildung C.1.3.: Übersicht über den verwendeten Wert des Parameters `adapt_delta` in der Simulation je Monat, Modell und Datensatz (eigene Darstellung)

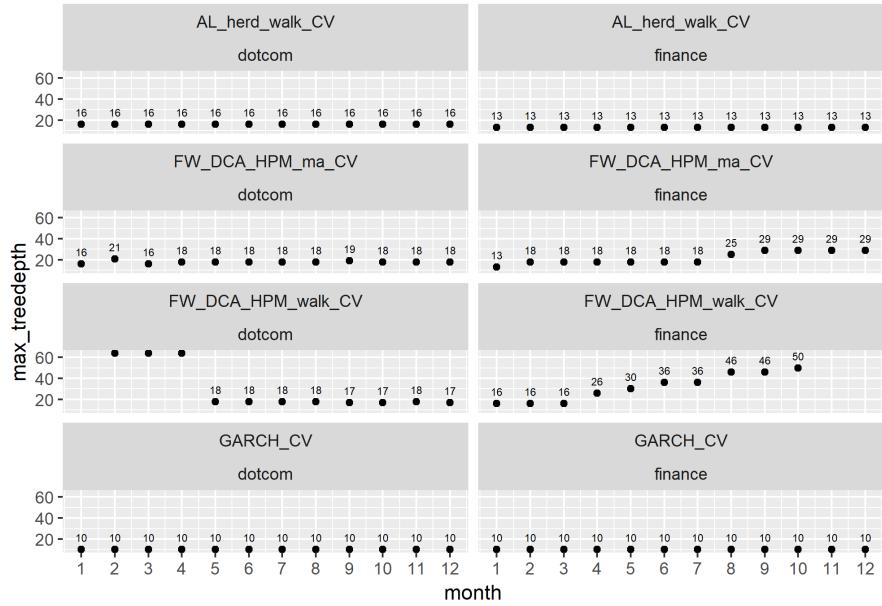


Abbildung C.1.4.: Übersicht über den verwendeten Wert des Parameters `max_treedepth` in der Simulation je Monat, Modell und Datensatz (eigene Darstellung)

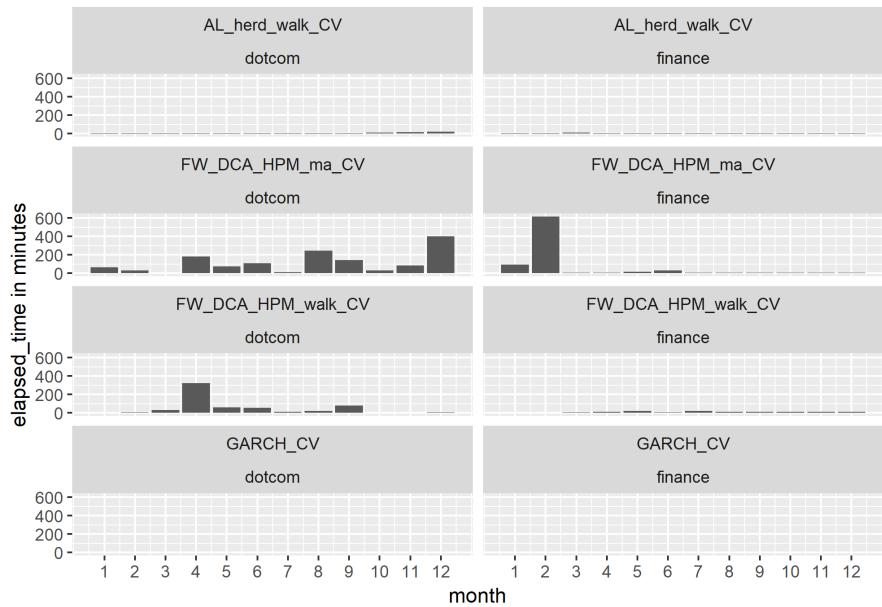


Abbildung C.1.5.: Übersicht über die durchschnittlich benötigte Zeit des Samplers zur Simulation je Monat, Modell und Datensatz (eigene Darstellung)

C.2. Preis-Vorhersagen

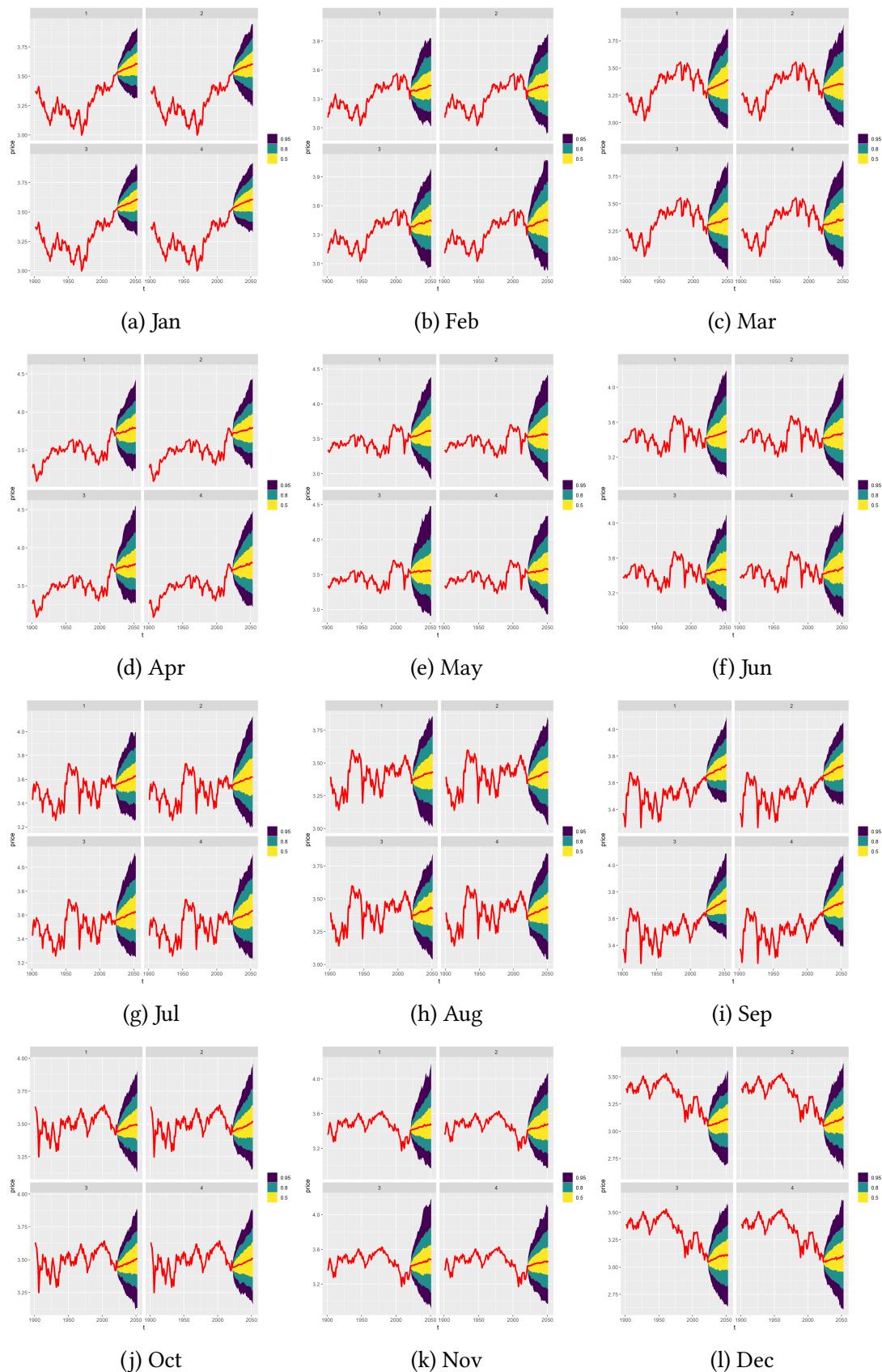


Abbildung C.2.1.: Preis-Vorhersagen des GARCH-Modells auf den Daten der Dotcom-Blase

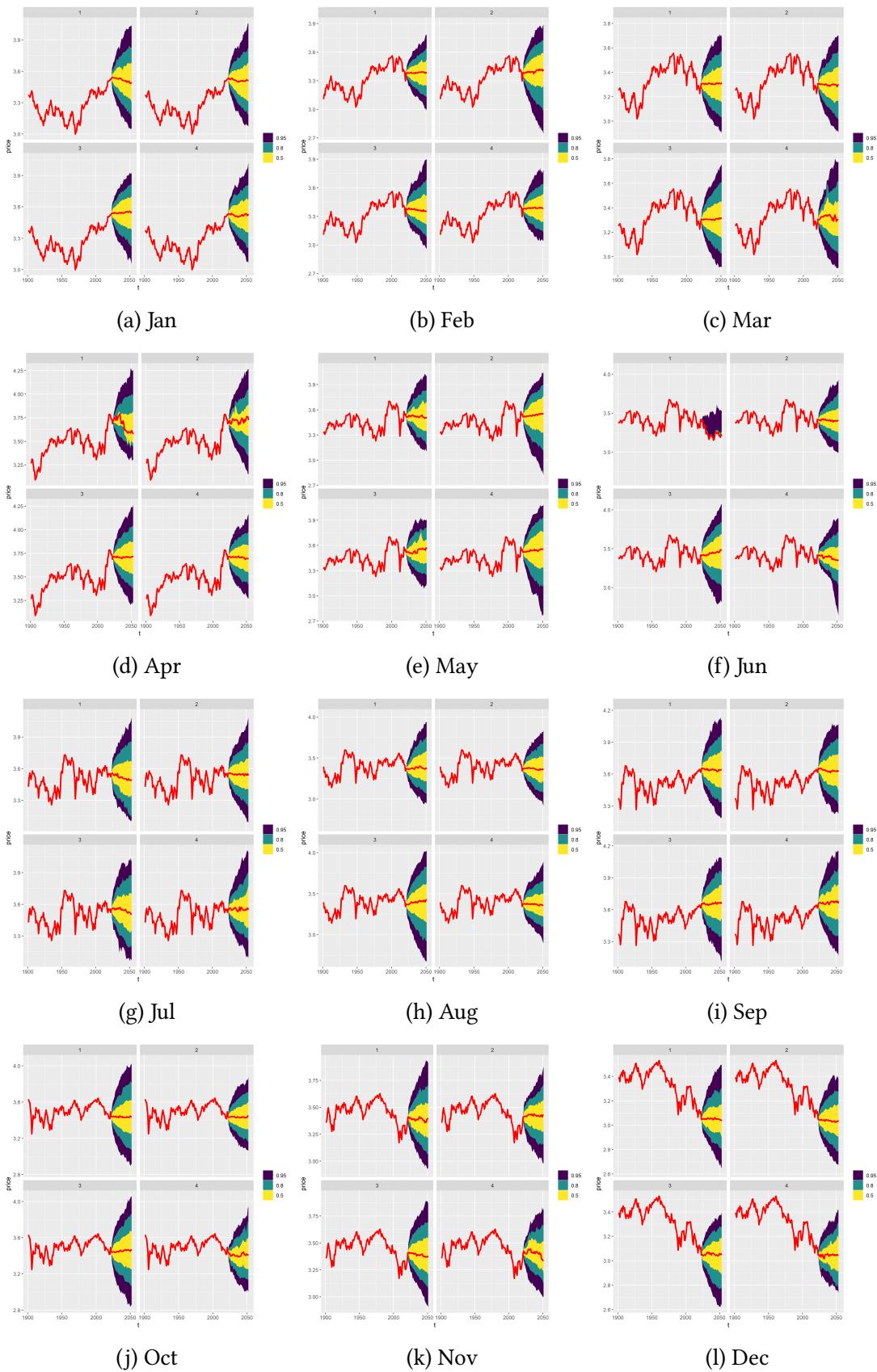


Abbildung C.2.2.: Preis-Vorhersagen des FW-Modells (ma) auf den Daten der Dotcom-Blase

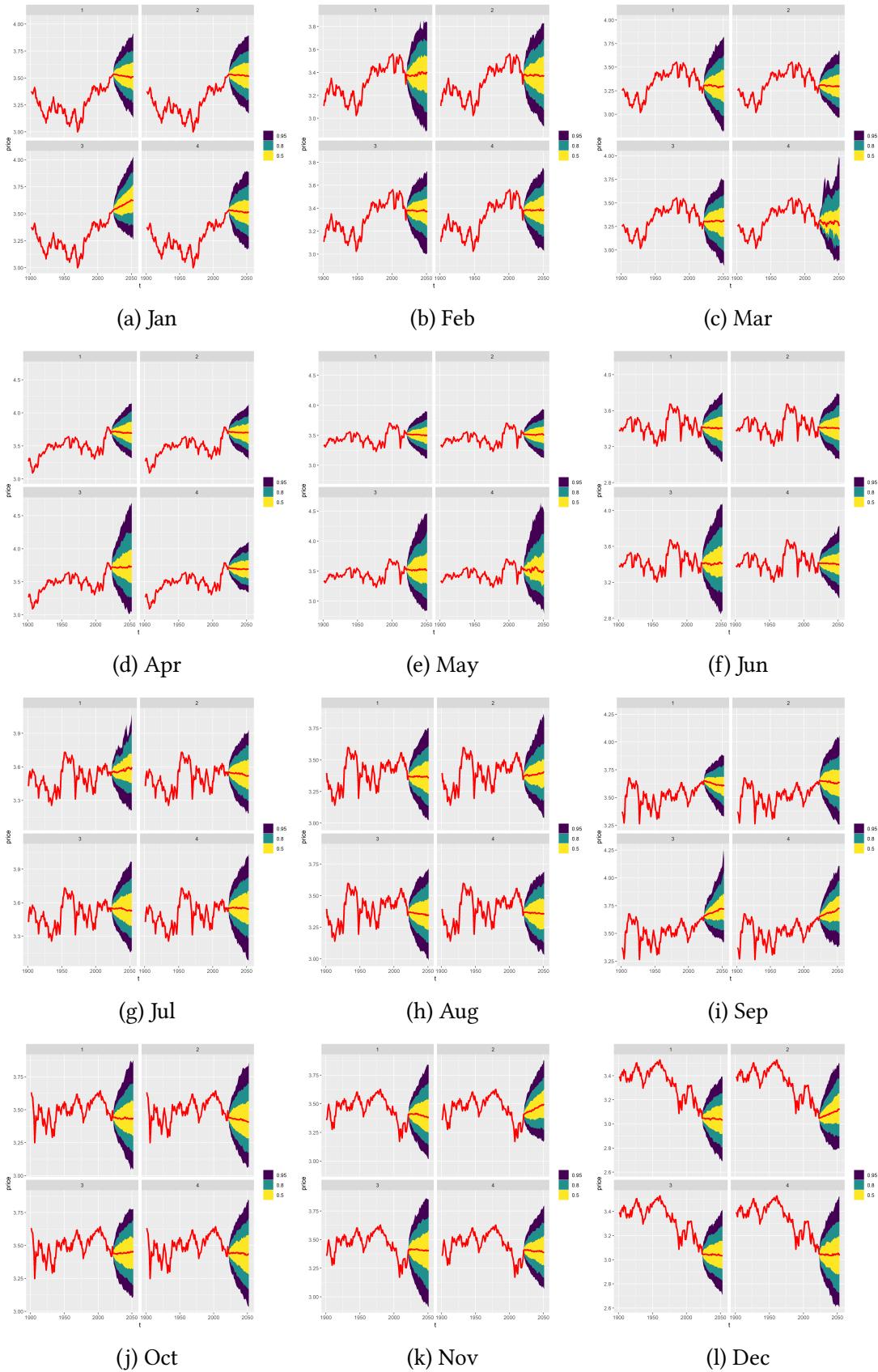


Abbildung C.2.3.: Preis-Vorhersagen des FW-Modells (walk) auf den Daten der Dotcom-Blase

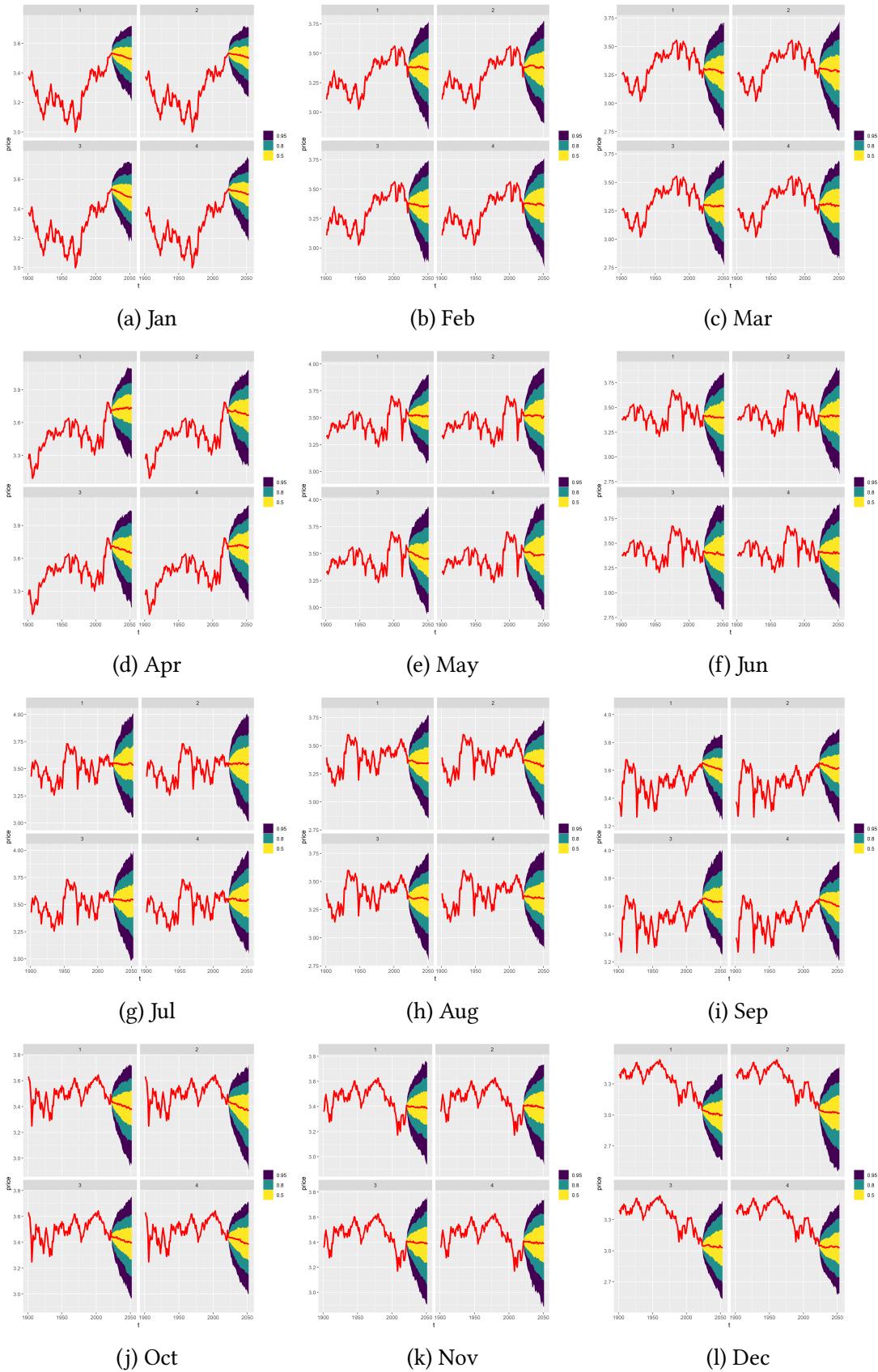


Abbildung C.2.4.: Preis-Vorhersagen des AL-Modells auf den Daten der Dotcom-Blase

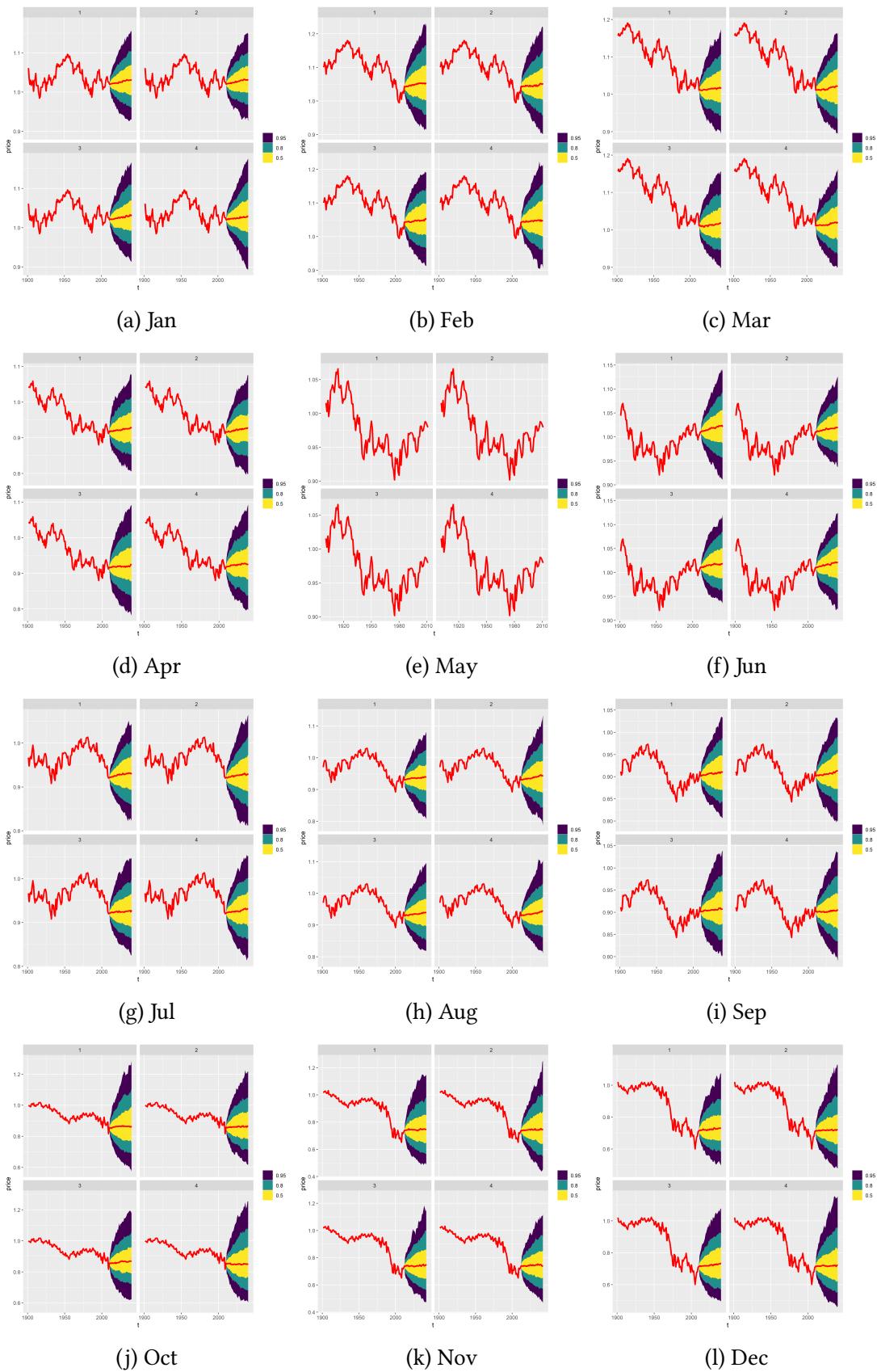


Abbildung C.2.5.: Preis-Vorhersagen des GARCH-Modells auf den Daten der Finanzkrise

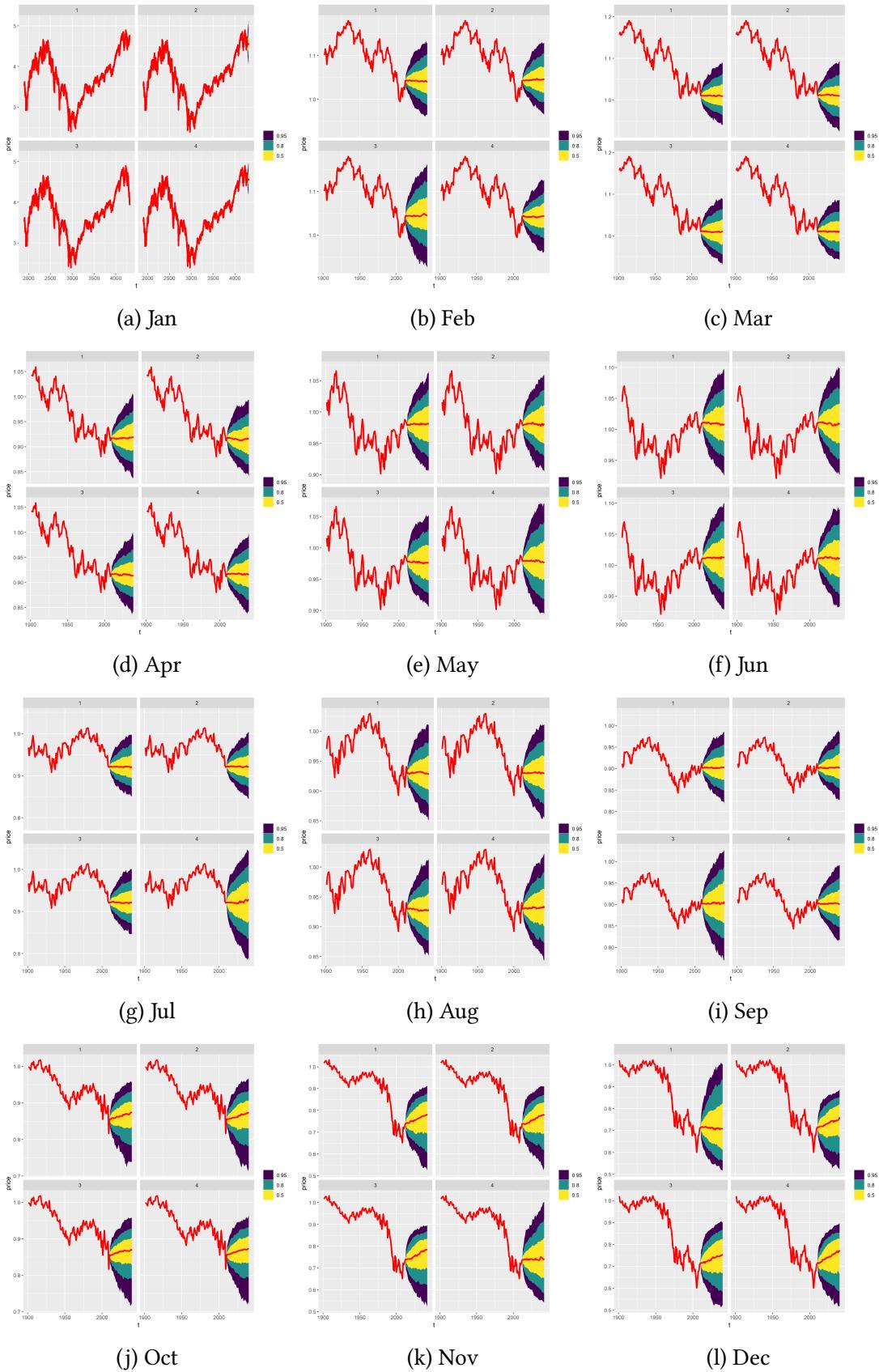


Abbildung C.2.6.: Preis-Vorhersagen des FW-Modells (ma) auf den Daten der Finanzkrise

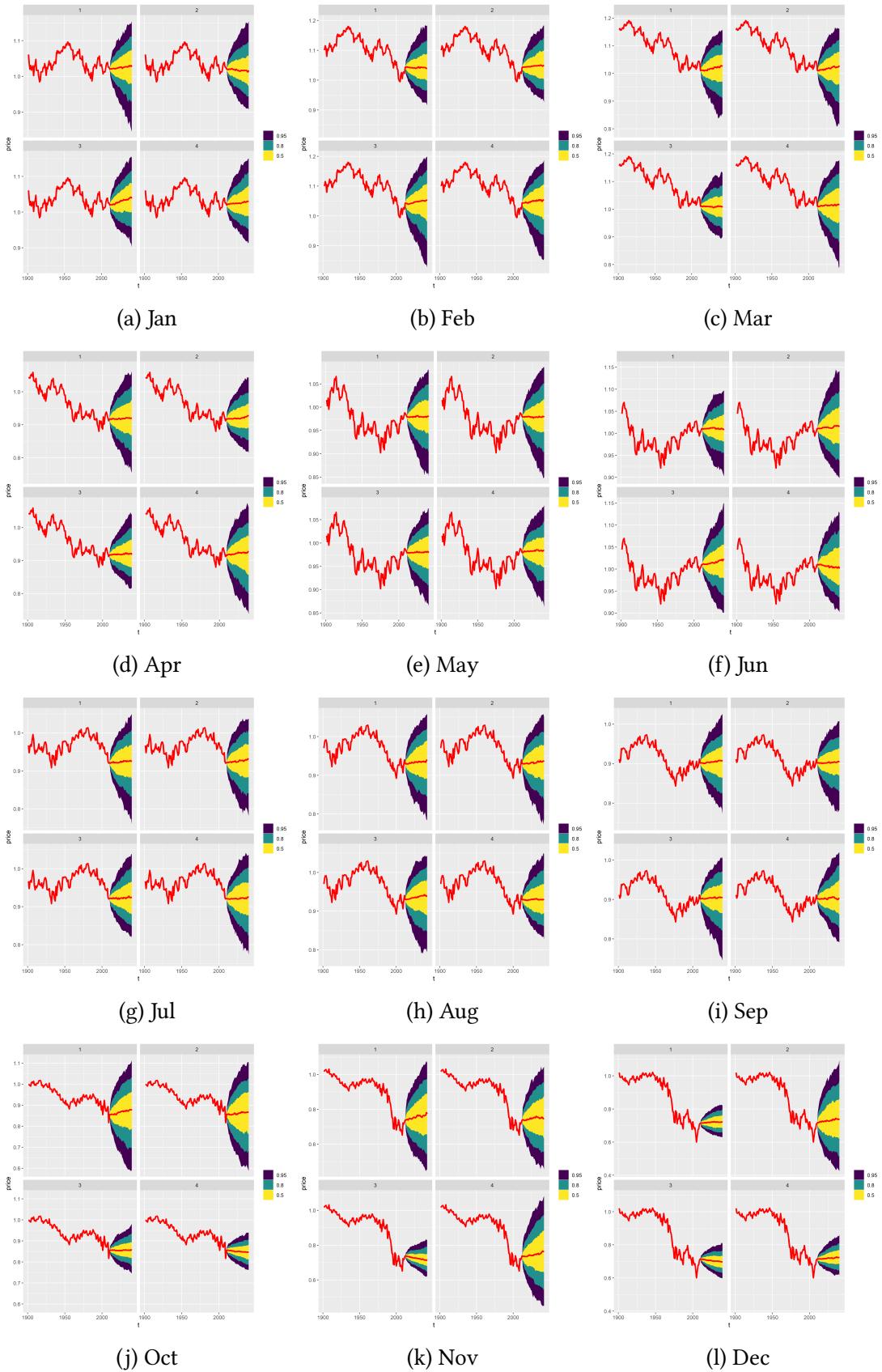


Abbildung C.2.7.: Preis-Vorhersagen des FW-Modells (walk) auf den Daten der Finanzkrise

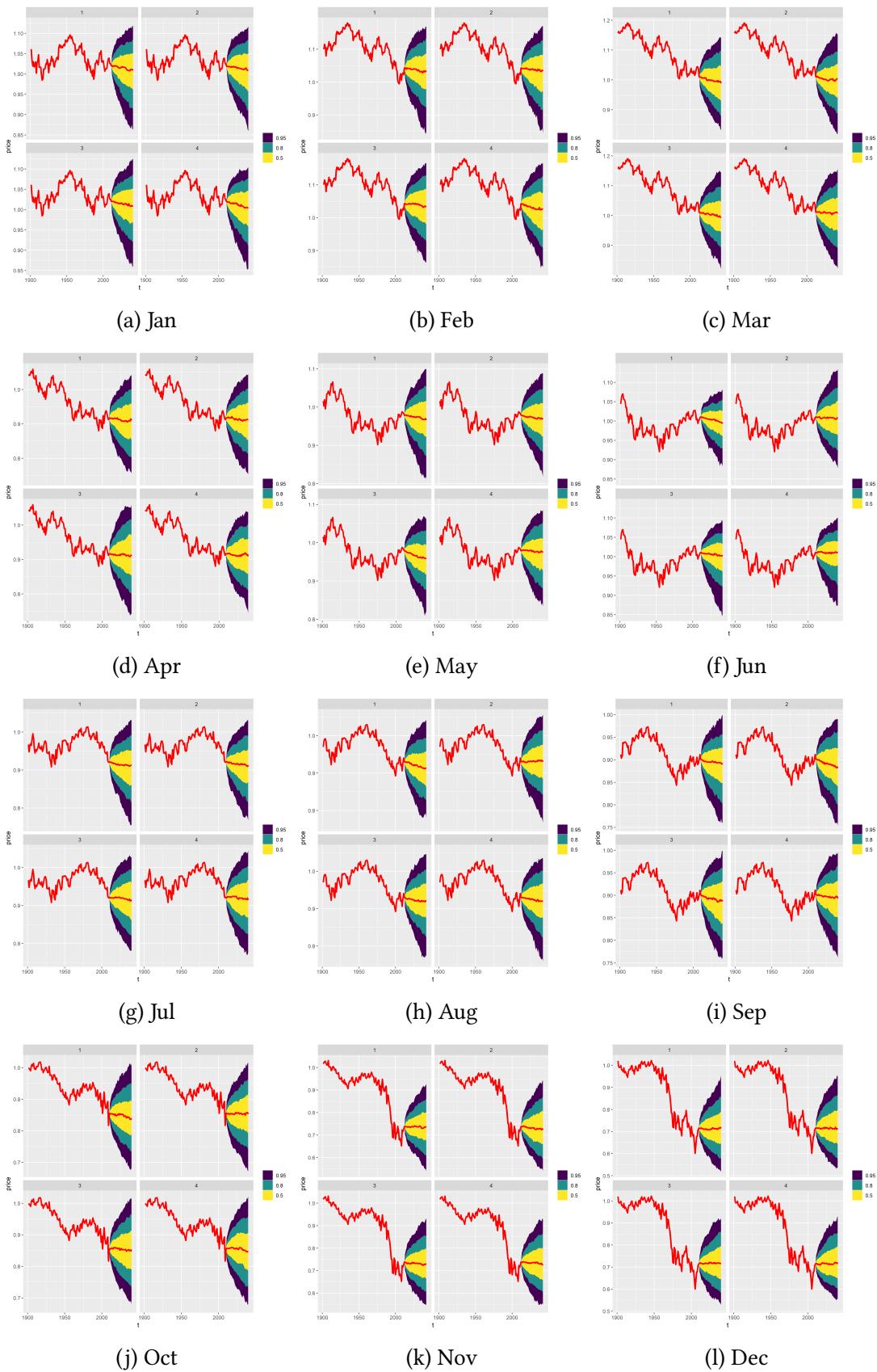


Abbildung C.2.8.: Preis-Vorhersagen des AL-Modells auf den Daten der Finanzkrise

C.3. Traceplots

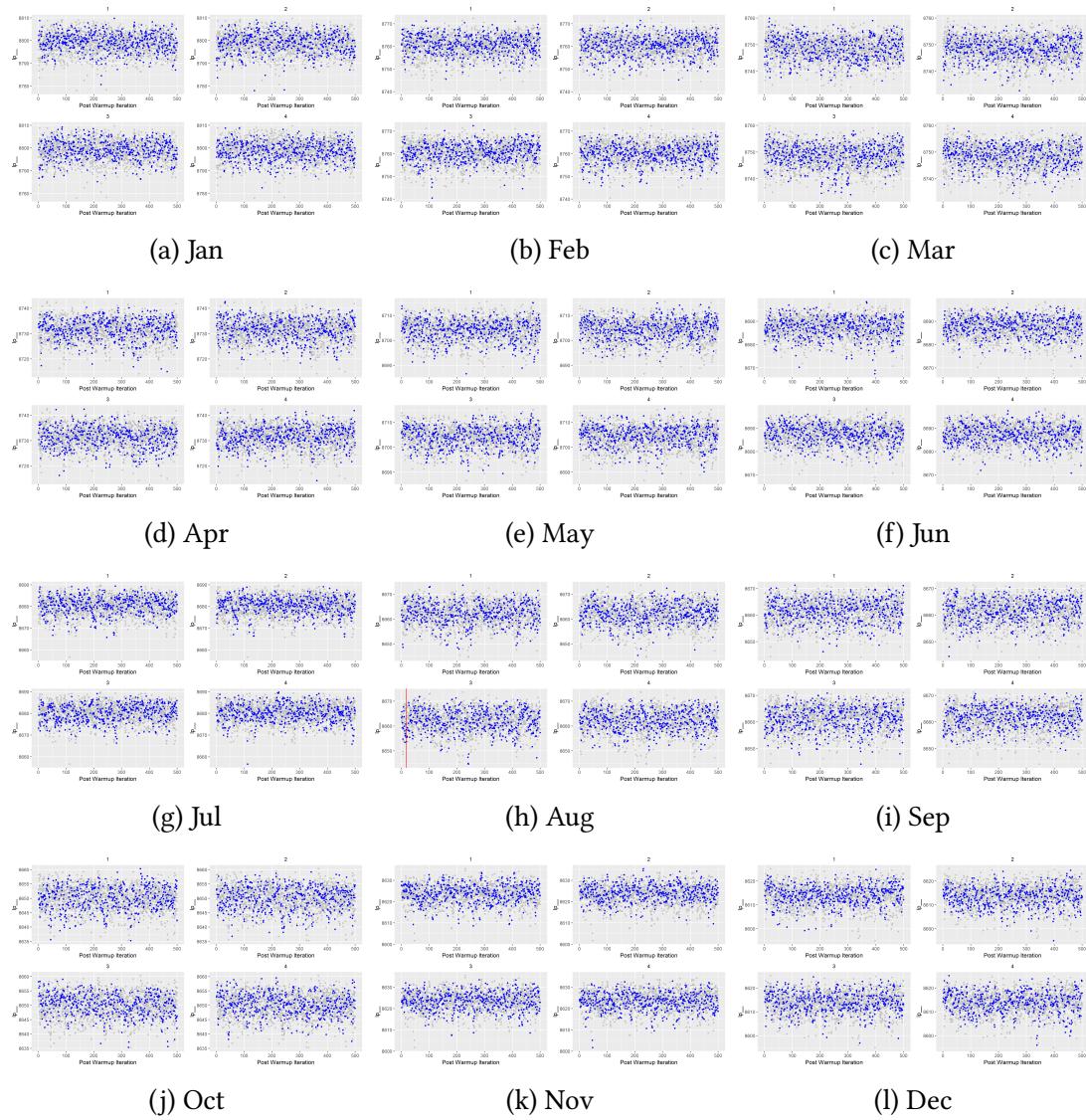


Abbildung C.3.1.: Preis-Vorhersagen des GARCH-Modells auf den Daten der Dotcom-Blase

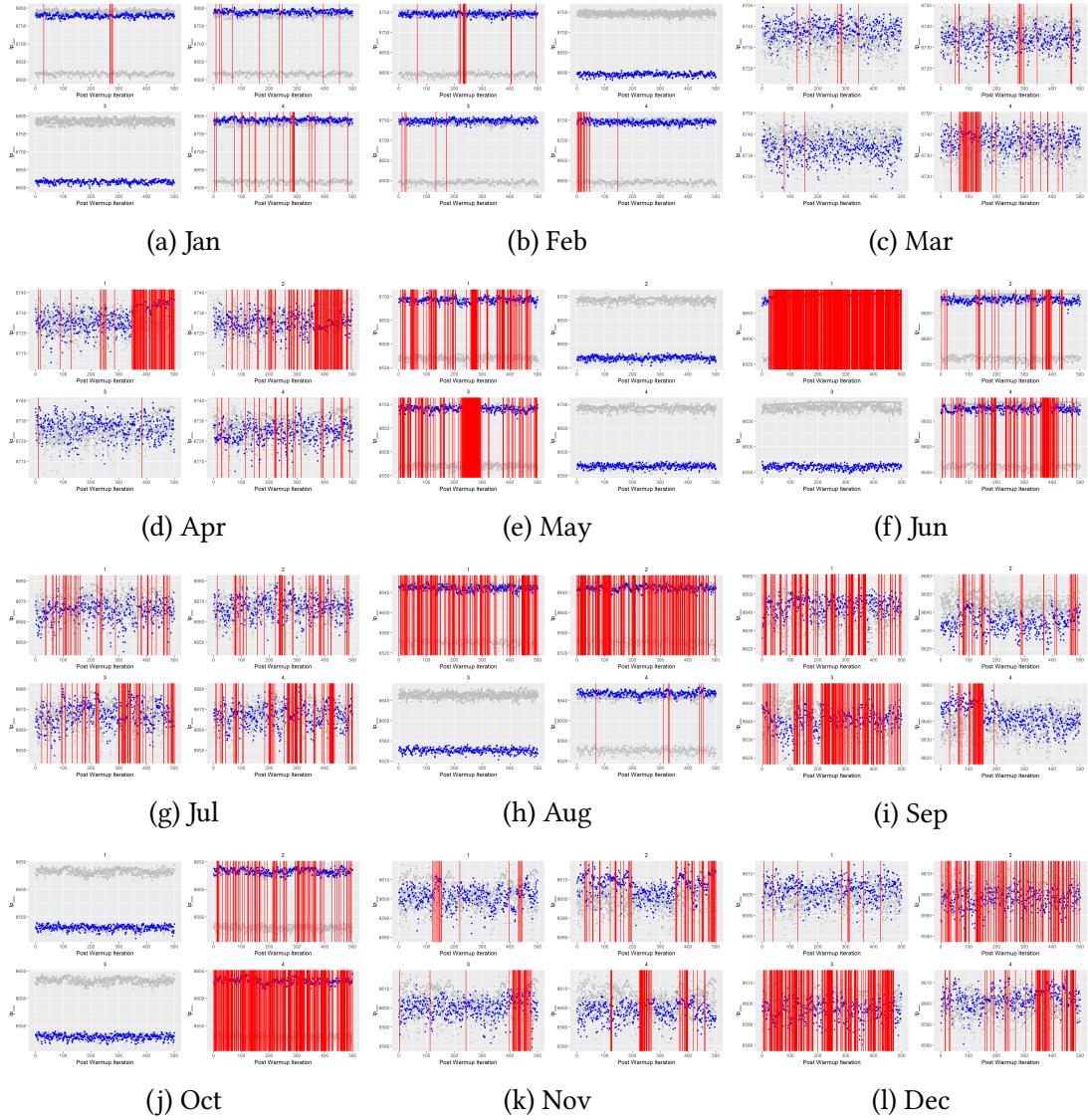


Abbildung C.3.2.: Preis-Vorhersagen des FW-Modells (ma) auf den Daten der Dotcom-Blase

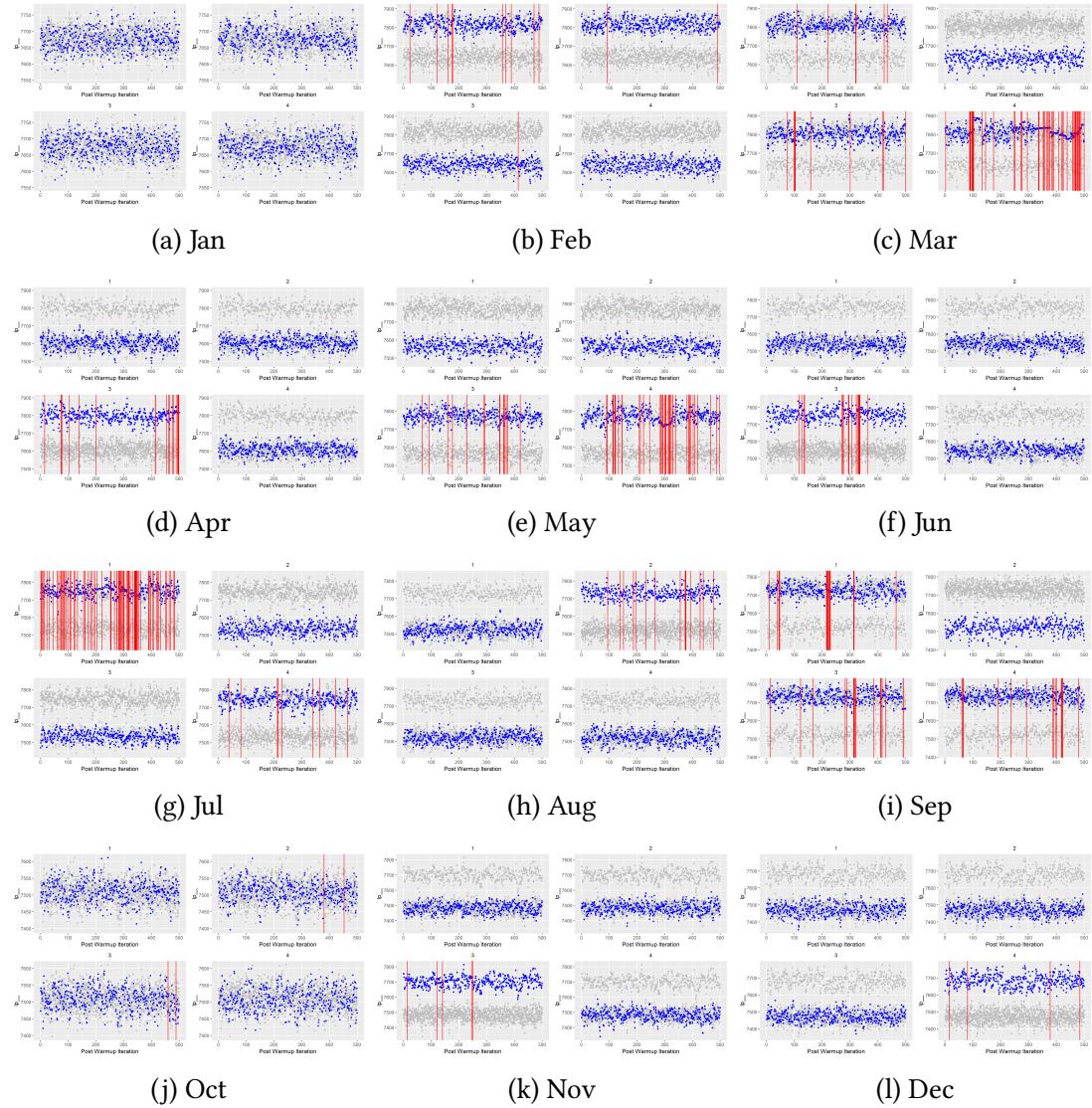


Abbildung C.3.3.: Preis-Vorhersagen des FW-Modells (walk) auf den Daten der Dotcom-Blase

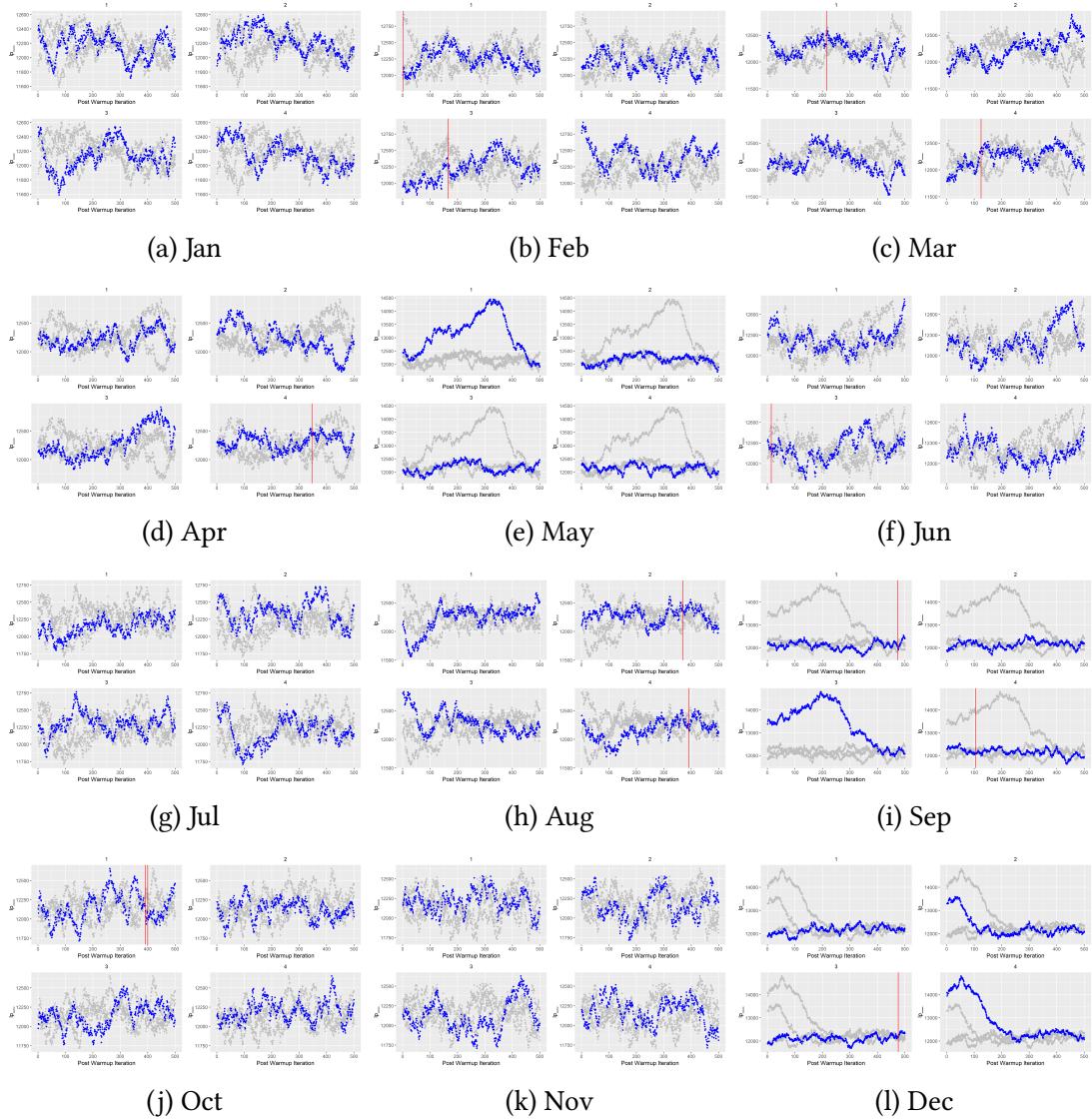


Abbildung C.3.4.: Preis-Vorhersagen des AL-Modells auf den Daten der Dotcom-Blase

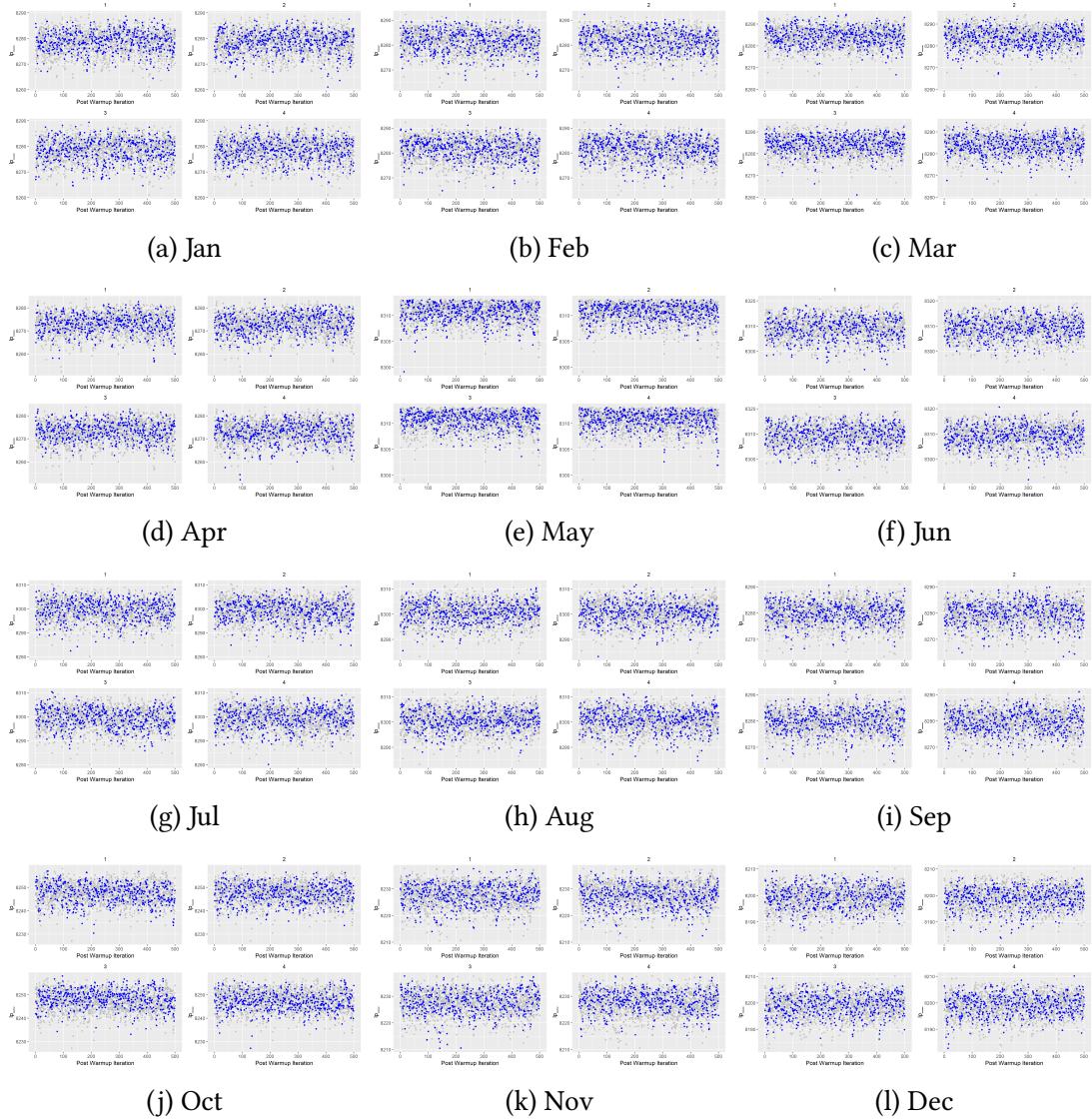


Abbildung C.3.5.: Preis-Vorhersagen des GARCH-Modells auf den Daten der Finanzkrise

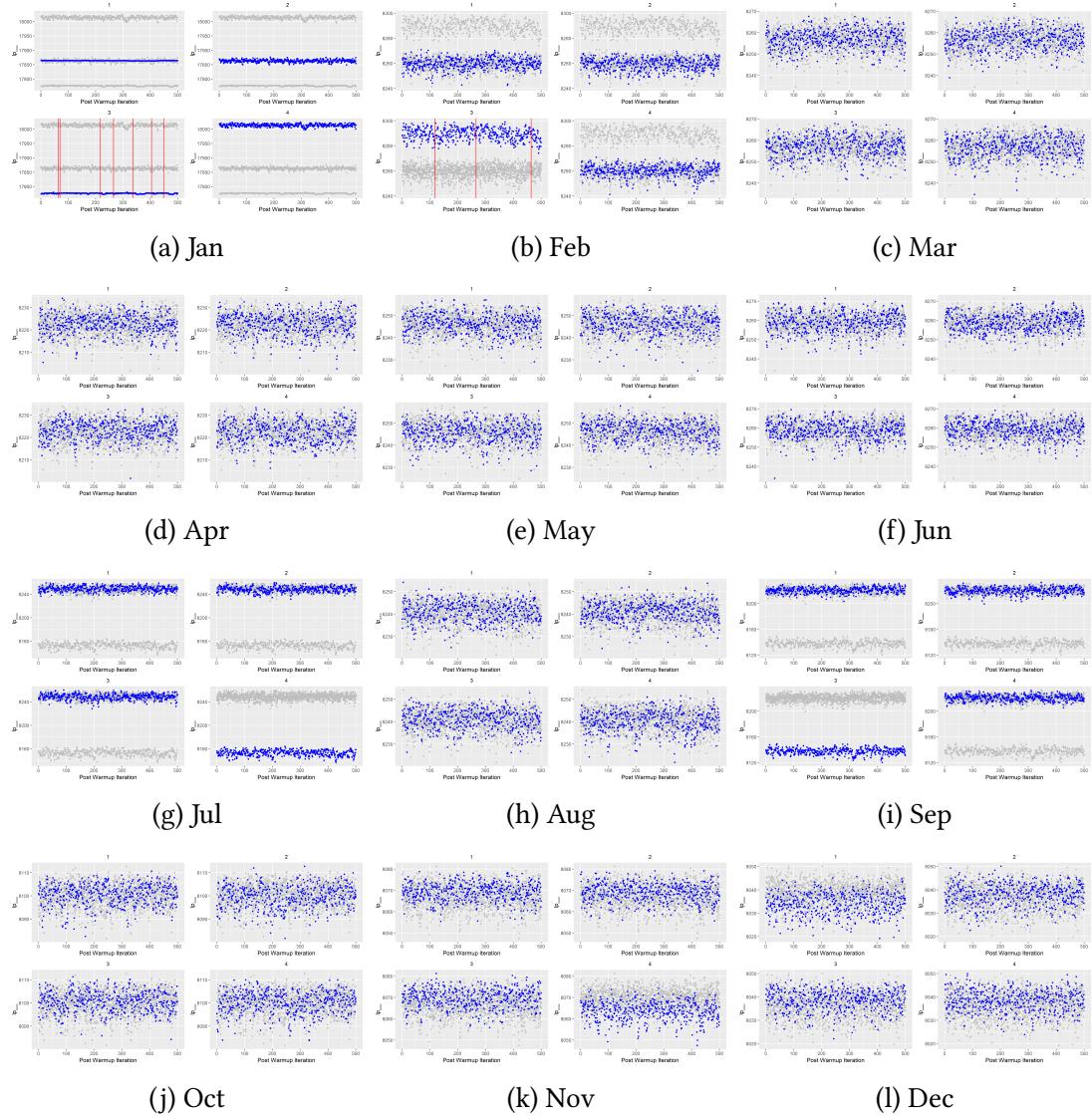


Abbildung C.3.6.: Preis-Vorhersagen des FW-Modells (ma) auf den Daten der Finanzkrise

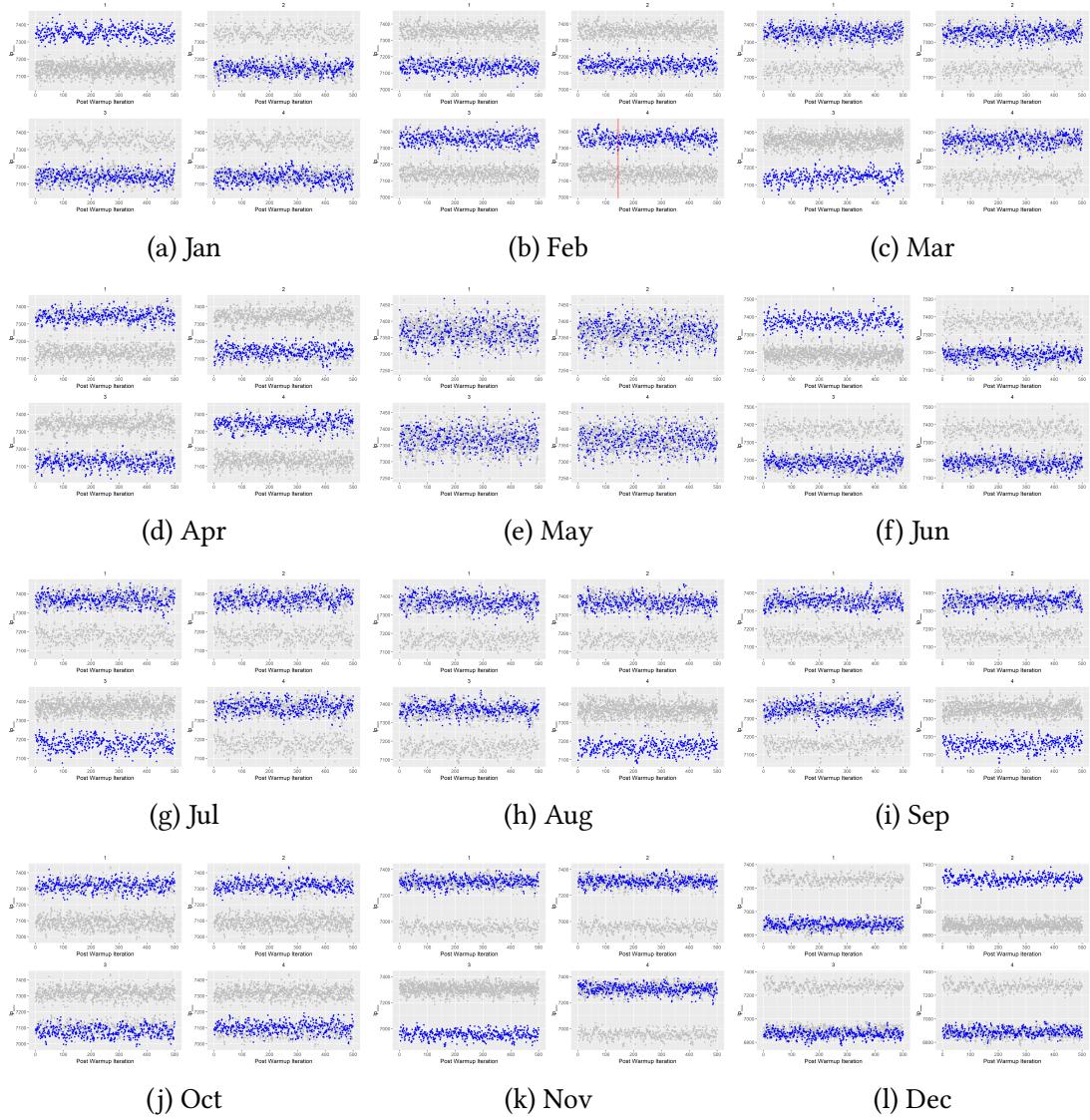


Abbildung C.3.7.: Preis-Vorhersagen des FW-Modells (walk) auf den Daten der Finanzkrise

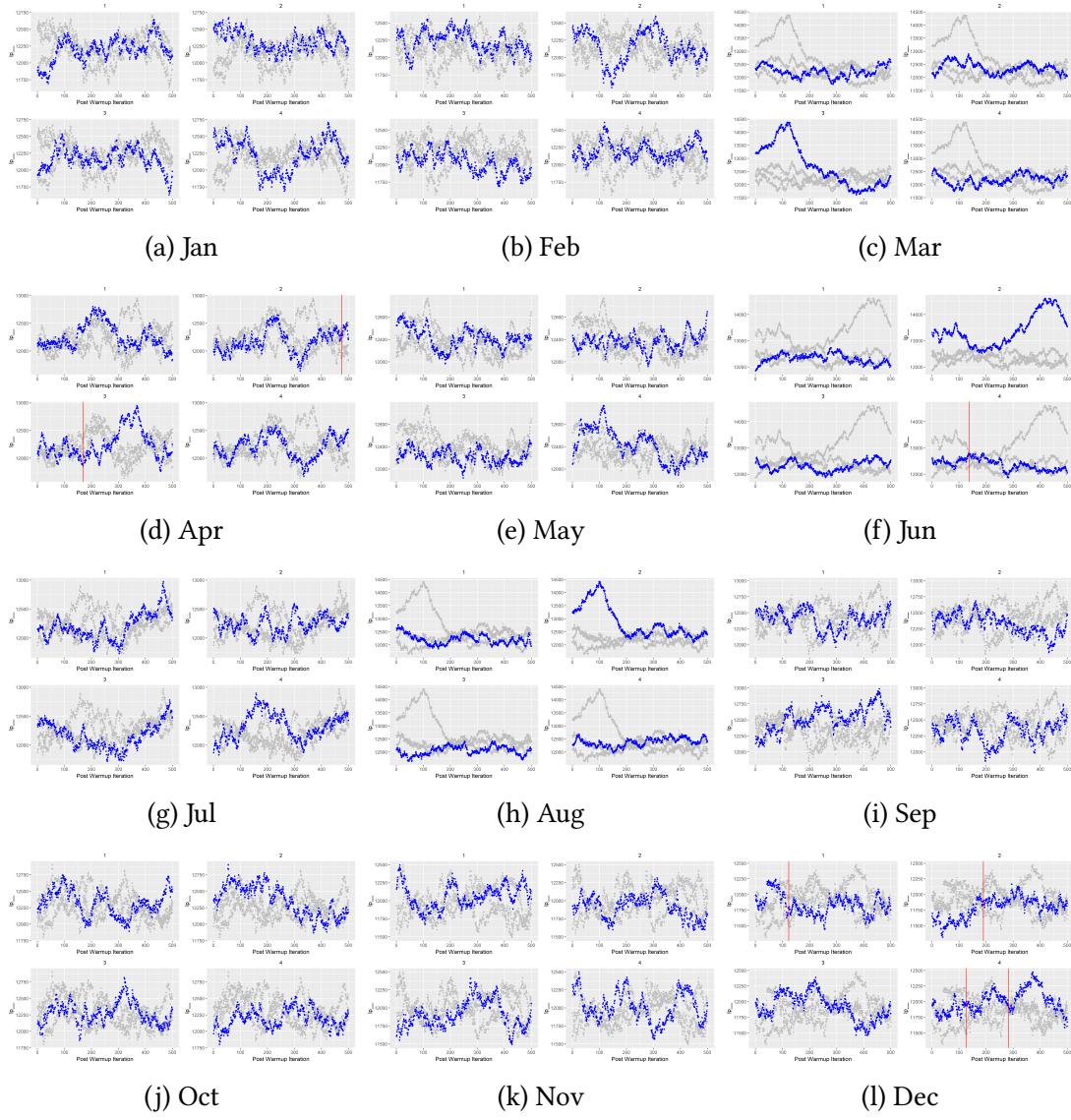


Abbildung C.3.8.: Preis-Vorhersagen des AL-Modells auf den Daten der Finanzkrise

Literatur

- [1] S. Alfarano, T. Lux und F. Wagner. „Time variation of higher moments in a financial market with heterogeneous agents: An analytical approach“. In: *Journal of Economic Dynamics and Control* 32.1 (2008), S. 101–136.
- [2] N. Bertschinger. „Notes on [1]“. 2019.
- [3] N. Bertschinger, I. Mozzhorin und S. Sinha. „Reality-check for Econophysics: Likelihood-based fitting of physics-inspired market models to empirical data“. In: *CoRR* abs/1803.03861 (März 2018). arXiv: 1803.03861. URL: <http://arxiv.org/abs/1803.03861>.
- [4] M. Betancourt. „A conceptual introduction to Hamiltonian Monte Carlo“. In: *arXiv preprint arXiv:1701.02434* (Jan. 2017).
- [5] T. Bollerslev. „Generalized autoregressive conditional heteroskedasticity“. In: *Journal of econometrics* 31.3 (1986), S. 307–327.
- [6] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li und A. Riddell. „Stan : A Probabilistic Programming Language“. In: *Journal of Statistical Software* 76.1 (Jan. 2017). DOI: 10.18637/jss.v076.i01.
- [7] R. F. Engle. „Autoregressive conditional heteroscedasticity with estimates of the variance of United Kingdom inflation“. In: *Econometrica: Journal of the Econometric Society* (1982), S. 987–1007.
- [8] *Fokker-Planck-Gleichung*. Dez. 2018. URL: <https://de.wikipedia.org/wiki/Fokker-Planck-Gleichung>.
- [9] R. Franke und F. Westerhoff. „Estimation of a structural stochastic volatility model of asset pricing“. In: *Computational Economics* 38.1 (2011), S. 53–83.
- [10] *Geometrische brownsche Bewegung*. Jan. 2018. URL: https://de.wikipedia.org/wiki/Geometrische_brownsche_Bewegung.
- [11] K. M. Hanson. „Markov chain Monte Carlo posterior sampling with the Hamiltonian method“. In: Bd. 4322. 2001, S. 4322 - 4322 –12. DOI: 10.1111/12.431119. URL: <https://doi.org/10.1111/12.431119>.
- [12] *Inverse-gamma distribution*. Feb. 2019. URL: https://en.wikipedia.org/wiki/Inverse-gamma_distribution.
- [13] Alan Kirman. „Ants, rationality, and recruitment“. In: *The Quarterly Journal of Economics* 108.1 (1993), S. 137–156.
- [14] *Logit*. Nov. 2018. URL: <https://de.wikipedia.org/wiki/Logit>.
- [15] *Mean-Reversion-Effekt*. Sep. 2017. URL: <https://de.wikipedia.org/wiki/Mean-Reversion-Effekt>.

- [16] *Modus*. Juli 2018. URL: [https://de.wikipedia.org/wiki/Modus_\(Statistik\)](https://de.wikipedia.org/wiki/Modus_(Statistik)).
- [17] *Simulated Annealing*. Jan. 2019. URL: https://de.wikipedia.org/wiki/Simulated_Annealing.
- [18] *Typical set*. Nov. 2018. URL: https://en.wikipedia.org/wiki/Typical_set.

Eigenständigkeitserklärung

gemäß Bachelor-Ordnung Informatik 2011 §25 Abs. 11

Hiermit erkläre ich Frau

Die vorliegende Arbeit habe ich selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst.

Frankfurt am Main, den _____

Unterschrift